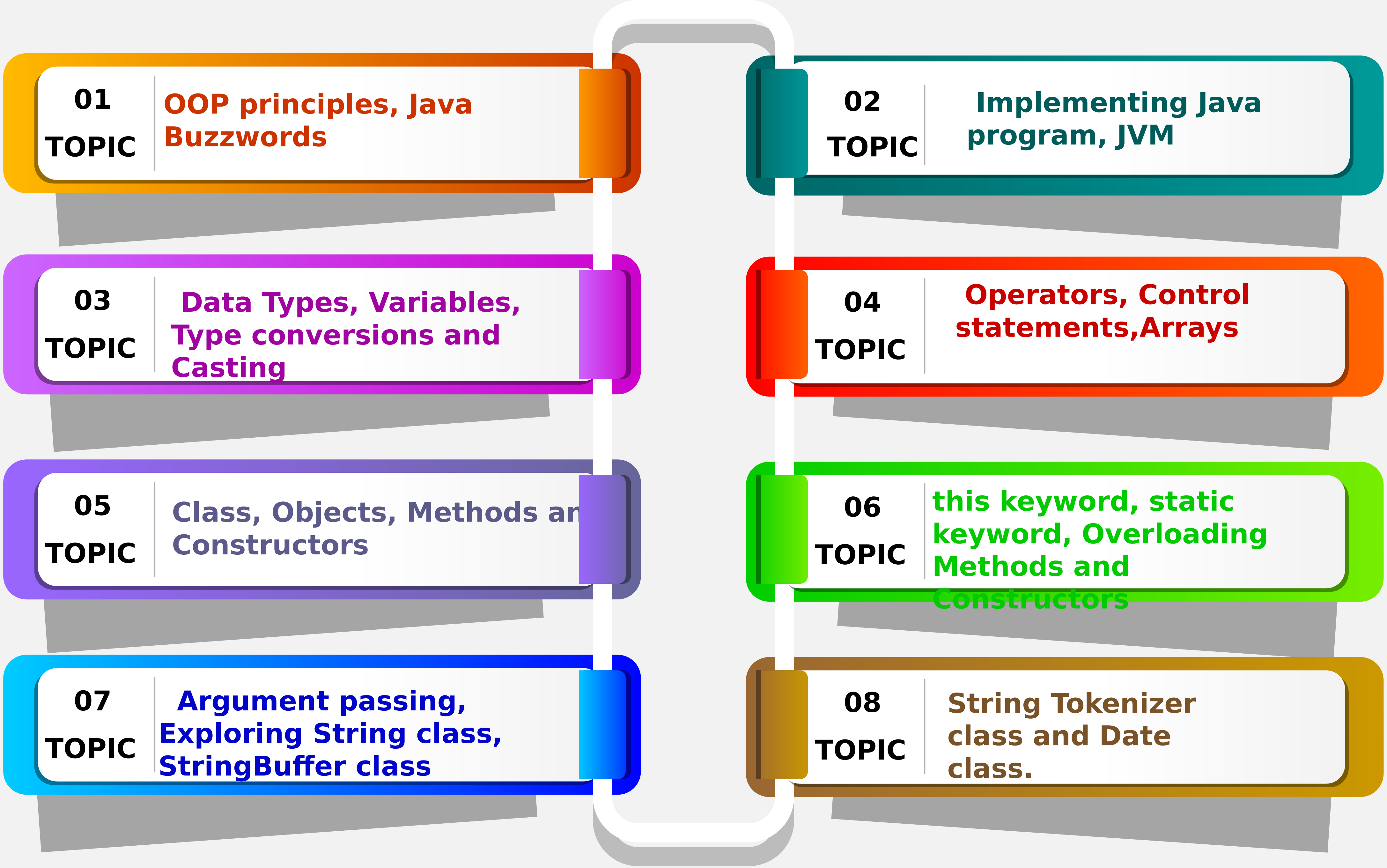
UNIT-1
A8601

O O P

OBJECT ORIENTED PROGRAMMING

PRESENTED BY

M.YGANDHAR
Department of IT
Vardhaman College of Engineering



01 TOPIC
OOP principles, Java Buzzwords

02 TOPIC
Implementing Java program, JVM

03 TOPIC
Data Types, Variables, Type conversions and Casting

04 TOPIC
Operators, Control statements,Arrays

05 TOPIC
Class, Objects, Methods an Constructors

06 TOPIC
this keyword, static keyword, Overloading Methods and Constructors

07 TOPIC
Argument passing, Exploring String class, StringBuffer class

08 TOPIC
String Tokenizer class and Date class.

Course Outcomes (COs)

- A8601.1 Make **use of various constructs** to **write** a console **application.**
- A8601.2 **Use principles of OOP** to **develop real time applications**.
- A8601.3 **Examine** the applications for **Exception Handling** and **Multithreading.**
- A8601.4 **Implement Collection** Framework **to organize data** efficiently .
- A8601.5 Build **GUI applications** using **AWT** and **Swings.**

What is Java?

- Java is a **programming language** and a **platform**.
- Java is a **high level**, **robust, secured** and **object-oriented programming language**.

Platform: Any **hardware** or **software environment** in which a **program runs** is **known as a platform**. Since **Java has its own runtime environment** (JRE) and **API**, it is called platform.

- Where it is used?
  - ✓ **Desktop Applications** such as acrobat reader, media player, antivirus etc.
  - ✓ **Web Applications** such as irctc.co.in, cleartrip.com etc.
  - ✓ **Enterprise Applications** such as banking,e-commerce applications.
  - ✓ **Mobile Applications** (J2ME)
  - ✓ **Embedded System**
  - ✓ **Smart Card**

Evolution of Java

- Currently, **Java is used in internet programming, mobile devices, games, e-business solutions** etc.

  i. **James Gosling**, Mike Sheridan, and Patrick Naughton **initiated** the **Java language** project in **June 1991**. The **small team** of **sun engineers** called **Green Team**.

  ii. **Originally designed** for **small, embedded systems** in electronic appliances **like set-top boxes.**

  iii. **Firstly, it was called "Greentalk"** by James Gosling and file extension was **.gt.**

  iv. **After that**, it was **called Oak** and was developed as a part of the Green project.

Why Oak?

Oak is a symbol of strength and choosen as a **national tree of many**

Why they choosed java name for java language?

vii. The **team gathered to choose a new name**. The **suggested words** were "**dynamic**", "**revolutionary**", "**Silk**", "**jolt", "DNA" etc**. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell and fun to say. **According to James Gosling "Java was one of the top choices along with Silk"**. Since **java was so unique**, most of the **team members preferred java.**

viii. Java is an **island of Indonesia** where **first coffee was produced (called java coffee).**

ix. Notice that **Java is just a name not an acronym.**

x. Originally **developed by James Gosling at Sun Microsystems** (which is now a subsidiary of Oracle Corporation) and **released in 1995.**

xi. In **1995, Time magazine** called **Java one of the Ten Best Products of 1995.**

xii. JDK 1.0 released in(January 23, 1996)

java versions

✓ There are **many java versions that have been released. Current stable release of Java is Java SE 8.**

1. **JDK Alpha and Beta (1995)**
2. **JDK 1.0 (23rd Jan, 1996)**
3. **JDK 1.1 (19th Feb, 1997)**
4. **J2SE 1.2 (8th Dec, 1998)**
5. **J2SE 1.3 (8th May, 2000)**
6. **J2SE 1.4 (6th Feb, 2002)**
7. **J2SE 5.0 (30th Sep, 2004)**
8. **Java SE 6 (11th Dec, 2006)**
9. **Java SE 7 (28th July, 2011)**
10. **Java SE 8 (18th March, 2014)**
11. **Java SE 9 (September, 21st 2017)**
12. **Java SE 10(March, 20th 2018)**
13. **Java SE 11 (18th March, 2014)**
14. **Java SE 12(March, 19th 2019)**
15. **Java SE 13(September, 17th 2019)**
16. **Java SE 14(March, 17th 2020)**
17. **Java SE 15(Expected in September 2020)**

OOPs Concepts / OOPs Principles

- **Object means** a **real word entity** such as **pen, chair, table etc.**
- **Object Oriented Programming** is a methodology or paradigm **to design** a **program using classes and objects.** It **simplifies** the **software development** and **maintenance** by **providing some concepts**:

   i. **Object**
   ii. **Class**
   iii. **Inheritance - OOP Principle -2**
   iv. **Polymorphism - OOP Principle -3**
   v. **Abstraction**
   vi. **Encapsulation - - OOP Principle -1**

i. **Object:** **Any entity** that **has state(properties)** and **behavior(perform some task)** is **known as an object.** **It is a real world entity.**

OOPs Concepts / OOPs Principles

iii. Inheritance :

- ✓ When **one object acquires all the properties** and **behaviors of parent object** i.e. known as **inheritance.** It **provides code reusability**. It is **used** to **achieve runtime polymorphism**.
- ✓ **Property transfer** from **Grand Parent** to **Parent** – to **children.**

iv. Polymorphism:

- ✓ When **one task is performed** by **different ways** i.e. known as **polymorphism.**
- ✓ **one in multiple forms** is known as **polymorphism.**

Example: When we are in **class – student** ,

When we are in **Market – Customer**,

When we are in **Home – Son / Daughter.**

- ✓ In java, we use **method overloading** and **method overriding** to achieve **polymorphism**.

OOPs Concepts / OOPs Principles

v. Abstraction:

- ✓ **Hiding internal details** and **showing functionality** is **known** as **abstraction.**

  Example: phone call, **we don't know the internal processing.**

- ✓ In java, **we use abstract class** and **interface to achieve abstraction.**
- ✓ **Exposing only the required essentials characteristics** & **behavior** with **respect to the context.**
- ✓ **Shows important things** to the **user** and **hide the internals details.**

  Example: ATM , BIKE

vi. Encapsulation:

- ✓ **Binding** (or wrapping) **code and data together into a single unit** is known as **encapsulation.**

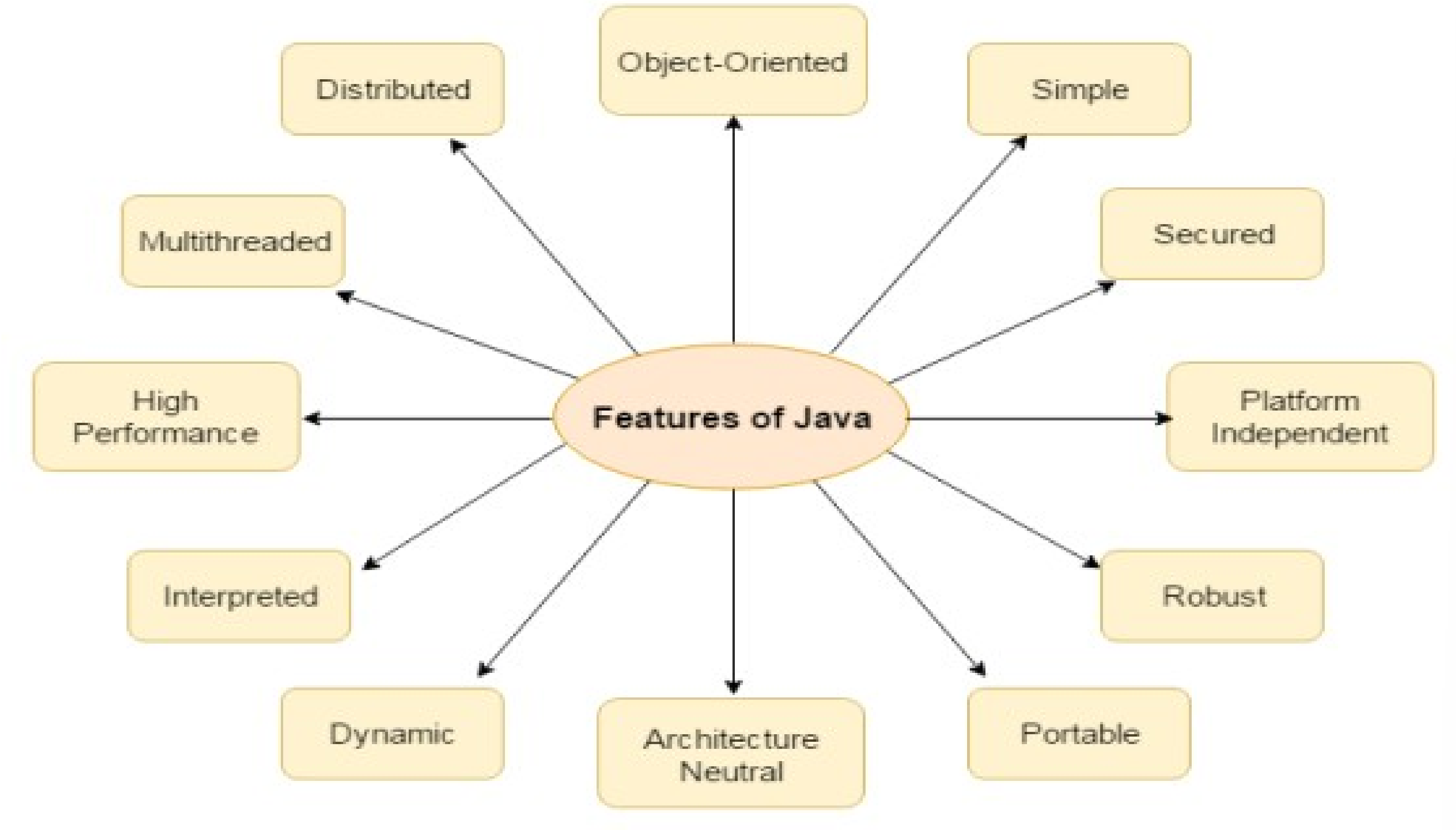
  Example: **Capsule,** it is wrapped with different medicines,

  Power Steering Car (Internally lot of Components tightly Coupled together)

- ✓ A **java class** is the **example of encapsulation**.

Features of Java / Java Buzz Word

- There is given many features of java. They are also known as java buzzwords. The Java Features given below are simple and easy to understand.



Features of Java / Java Buzz Worc

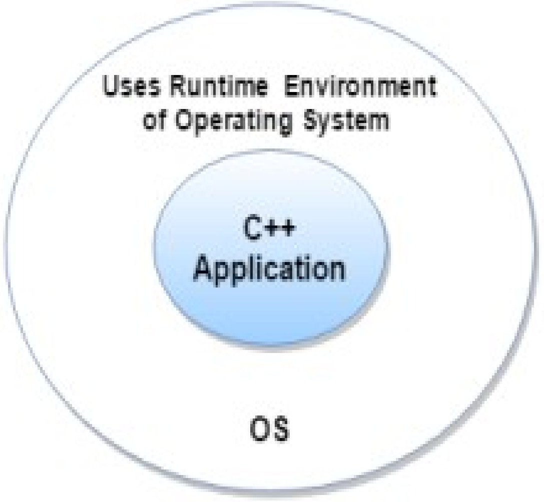
- Object Oriented: In Java, **everything is an Object**. Java can be **easily extended** since it is **based on the Object model.**
- Simple: **Java is designed** to be **easy to learn**. If you **understand the basic concept of OOP Java** would be **easy to master.**
- Secure: With **Java's secure feature it enables to develop virus-free applications**. **Authentication** techniques are based on **public-key encryption.**
- Platform independent: **Unlike** many other programming languages including **C and C++,** when **Java** is compiled, it is **not compiled into platform specific machine**, rather into platform independent byte code. Robust: Java makes **an effort to eliminate error prone situations** by emphasizing **mainly on compile time error checking** and **runtime checking.**

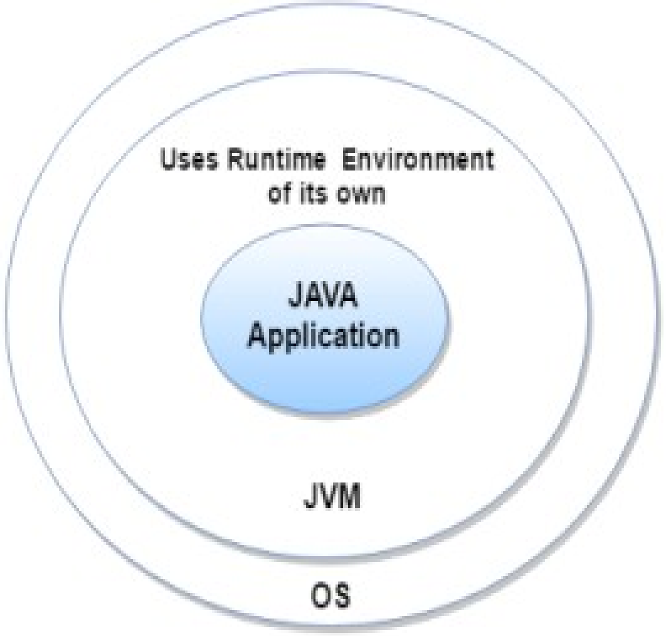
Features of Java / Java Buzz Worc

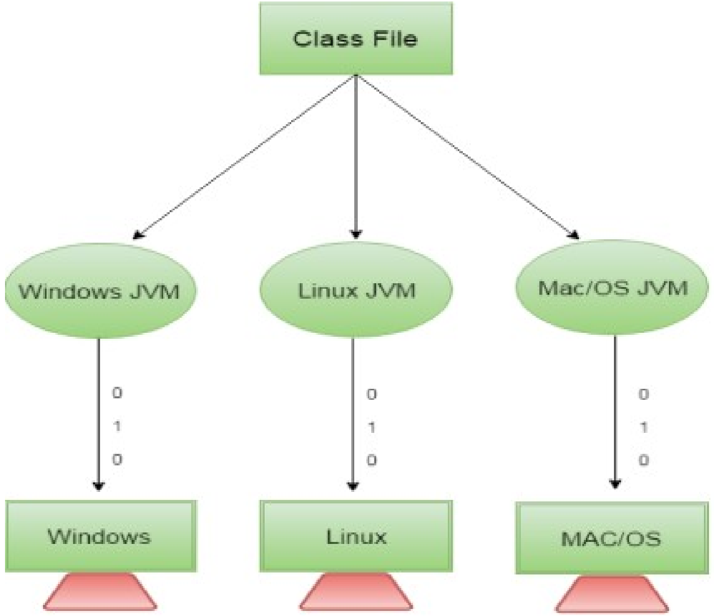
- Portable: **Being architectural-neutral** and **having no implementation dependent .**We may carry the **java bytecode** to **any platform.**
- Architectural-neutral: Java **compiler generates** an **architecture-neutral object file** format which makes the **compiled code to be executable on many processors**, with the presence of Java runtime system.
- Dynamic: Java is considered to **be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs** can **carry** extensive amount of **runtime information** that can be **used** to **verify** and resolve **accesses** to **objects** on run-time.
- Interpreted: **Java byte code is translated to native machine instructions** and is not stored anywhere.
- High Performance: With the **use of Just-In-Time compilers, Java enables high**

Features of Java / Java Buzz Worc

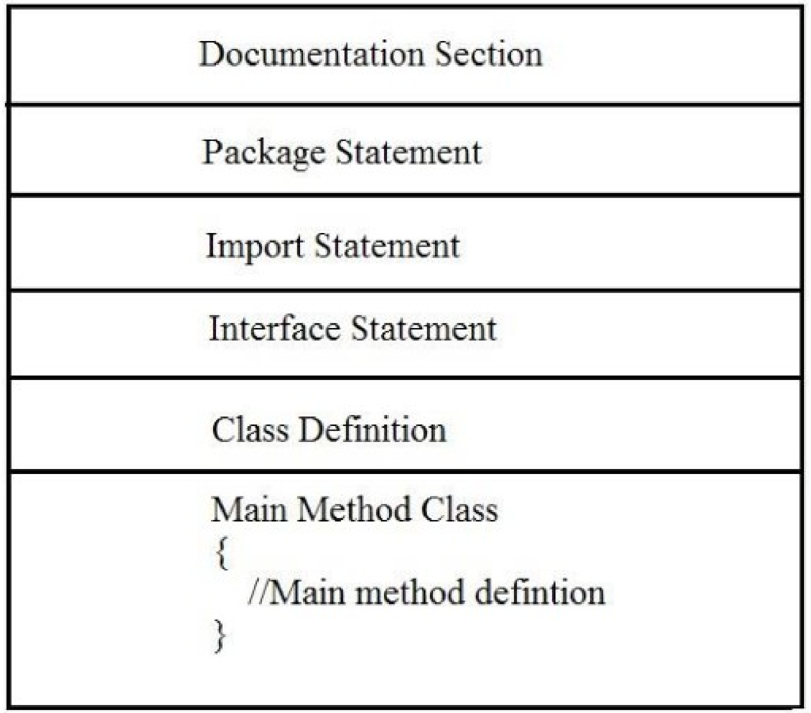
- Multithreaded: With Java's multithreaded feature it is **possible to write programs that can do many tasks simultaneously.** This feature allows developers to construct smoothly running interactive applications.
- Distributed: We can **create distributed applications in java. RMI and EJB are used for creating distributed applications.** We may **access files by calling the**







Basic Structure of Java Program



Documentation Section

Package Statement

Import Statement

Interface Statement

Class Definition

```
Main Method Class
{
   //Main method defintion
}
```

```
public class Simple
{
        public static void main( String args[ ] )
        {
              System.out.println("Welcome To the World of
        Java");
        }
}
```

save this file as Simple.java

To compile: javac

Simple.java

Basic Structure of Java Program

i. **class keyword** is **used** to **declare a class** in **java.**

ii. **public keyword** is an **access modifier** which **represents visibility**, it means it is **visible to all.**

iii. **static** is a **keyword**, if we **declare any method as static**, it is known as static method. The core **advantage of static** method is that **there is no need to create object to invoke or call** the **static method.**

iv. **void** is the **return type of the method**, it means it **doesn't return any value**.

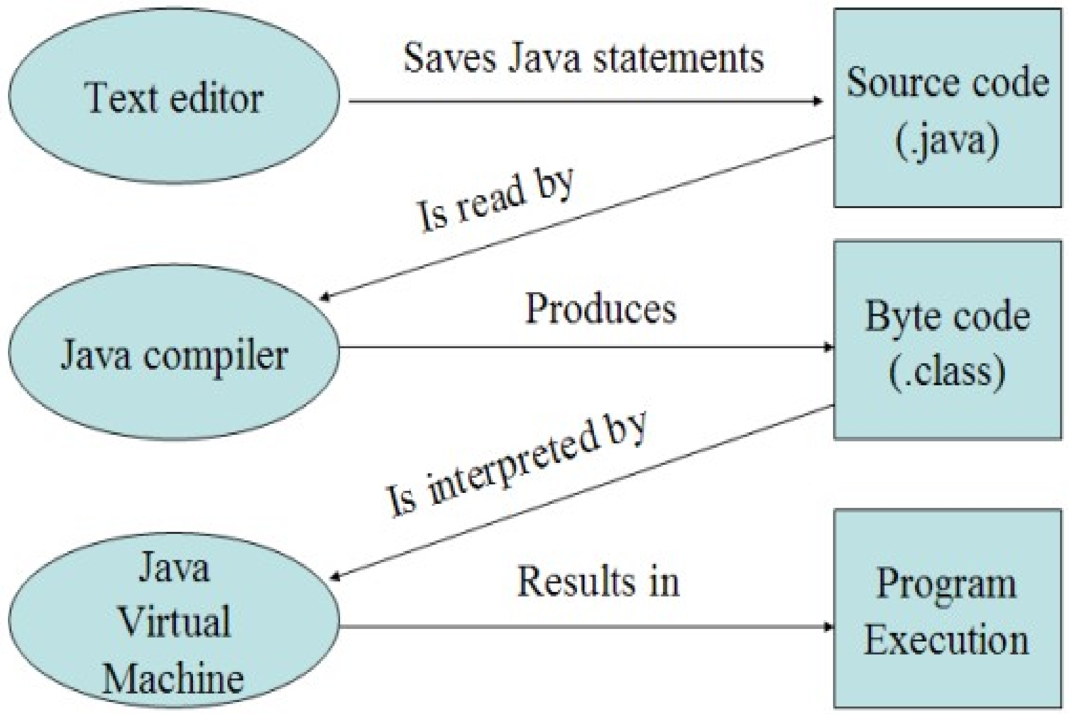
v. The **main method** is **executed by the JVM**, so **it doesn't require** to **create object** to **invoke the main method**. It **represents startup of the program**.

vi. **String[] args** is **used** for **command line argument.**

vii. **System.out.println()** is **used print** statement.

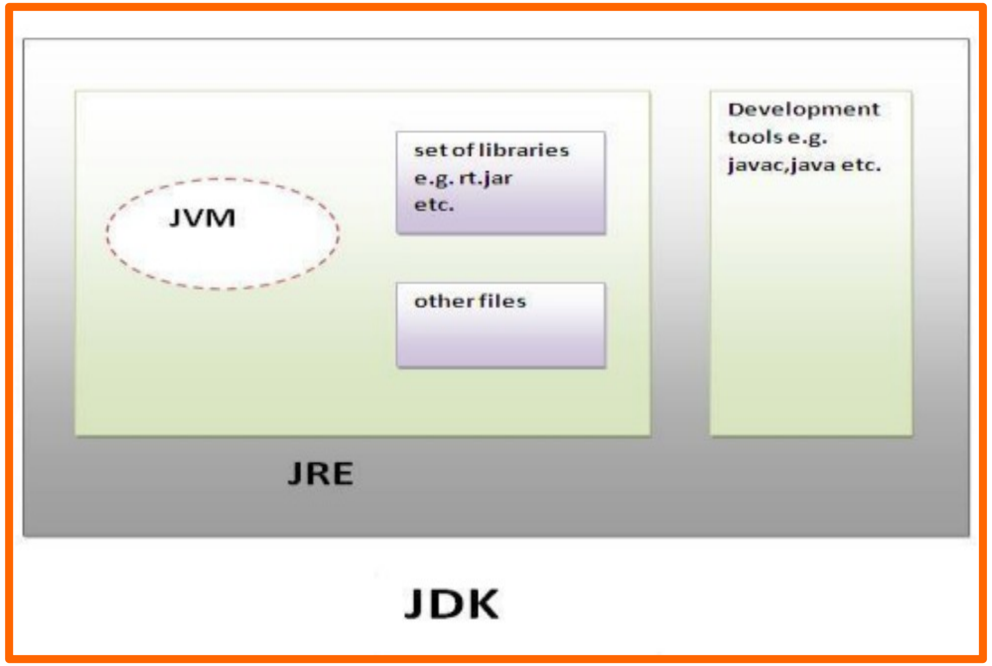
- ➢ **System** is a **class** Which is **present in java.lang package**
- ➢ **Out** is a **static final field (variable)** of **Printstream class**

Java program execution



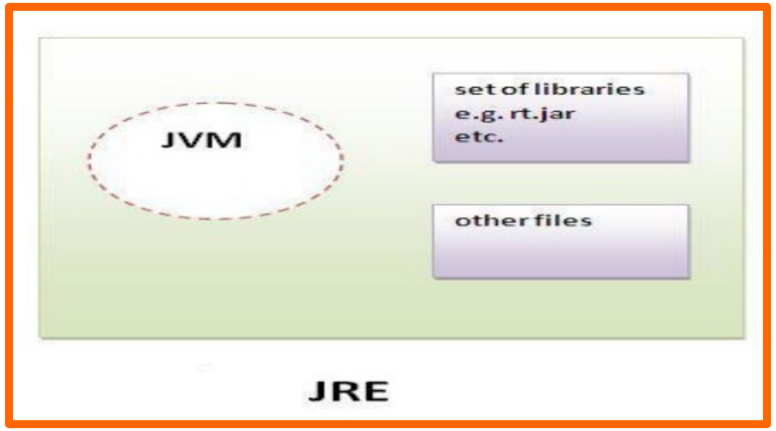
JDK (Java Development Kit)

- JDK is an acronym for It **physically exists**.
- It contains **JRE + development tools.**

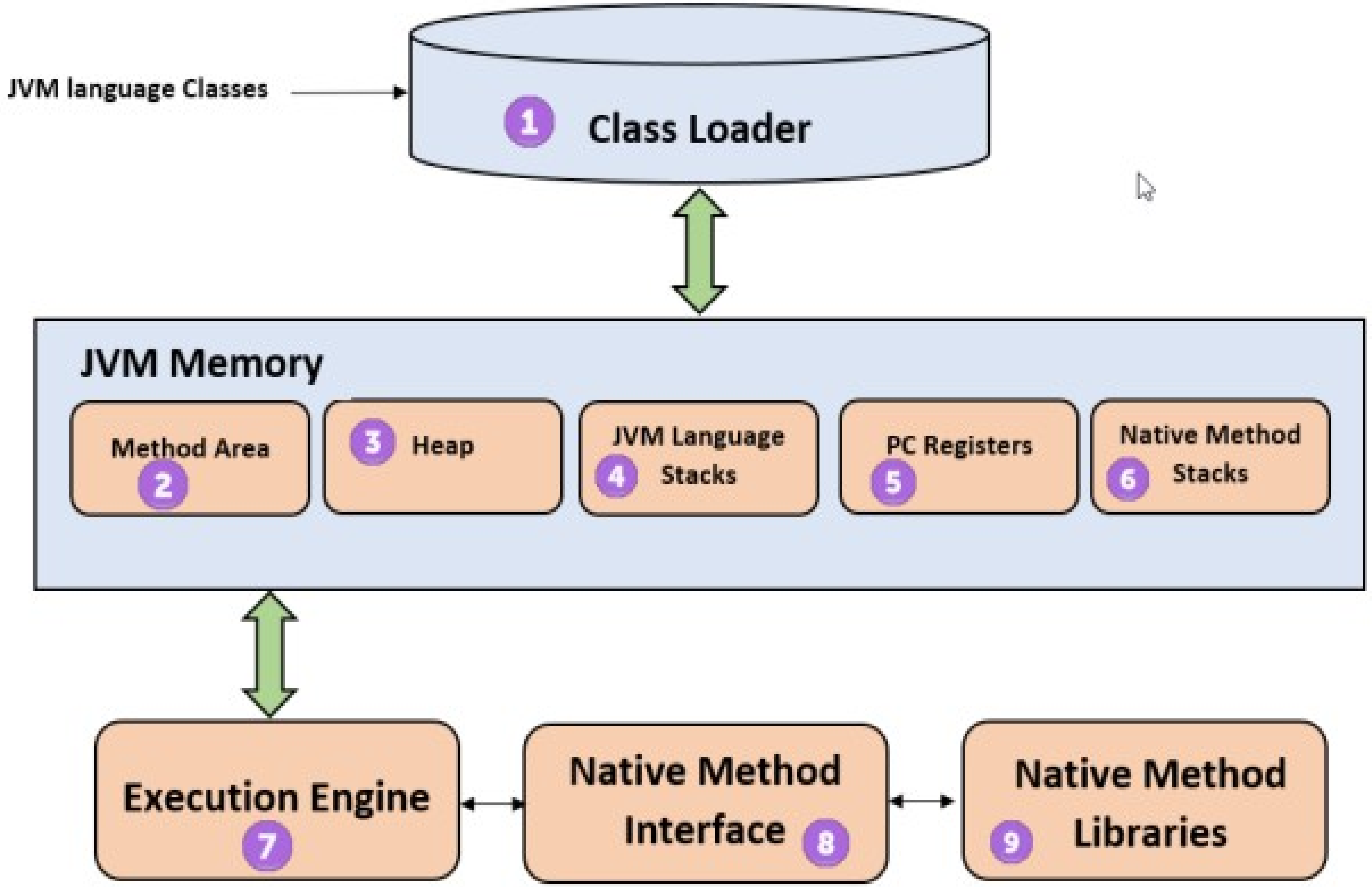

JVM
set of libraries e.g. rt.jar etc.
other files
Development tools e.g. javac,java etc.
JRE
JDK

JRE (Java Runtime Environment)

- It is used to provide **runtime environment.**
- It is the implementation of **JVM.**
- It contains **set of libraries + other files that JVM** uses **at runtime.**
- **Implementation** of **JVMs** is also actively **released by other companies** besides Sun Micro Systems.



JVM (Java Virtual Machine)



JVM (Java Virtual Machine)

- JVM (Java Virtual Machine) **provides runtime environment** in which **java byte code** can be **executed.**
- The JVM performs following operation:
  - **Loads code**
  - **Verifies code**
  - **Executes code**
  - **Provides runtime environment**

1) Class loader:

- ✓ It is a **subsystem of JVM** that is **used to load class files.**
- ✓ It **loads, links** and **initializes** the **class file** when it refers to a class for the **first time at runtime.**

2) Method Area:

- ✓ It is the **class-level data** will be **stored here**, including static **variables**.
- ✓ There is **only one method** area **per JVM**, and it is a **shared resource**

JVM (Java Virtual Machine)

3) Heap:

- ✓ It is the **runtime data area** in which **objects are allocated**.
- ✓ **All the Objects** and their **corresponding instance variables** and **arrays will be stored here.**

4) Stack:

- ✓ For **every thread**, a **separate runtime stack** will be **created.**
- ✓ For **every method call**, **one entry will be made** in the **stack memory** which is called Stack Frame.
- ✓ All **local variables** will **be created in the stack memory**.

5) Program Counter Register :

- ✓ **Each thread will have separate PC Registers**, to **hold** the **address of current executing instruction** once the instruction is executed the **PC register will be updated with the next instruction.**

JVM (Java Virtual Machine)

7) Execution Engine :

- ✓ The Execution Engine **reads the bytecode** and **executes it piece by piece.**
- ✓ **It contains three components:**

7.1 Interpreter:

- ✓ stream
- ✓ The **interpreter interprets** the **bytecode faster** but **executes slowly**.
- ✓ The **disadvantage** of the **interpreter** is that when **one method** is **called multiple times**, every time a **new interpretation is required.**

7.2 Just-In-Time (JIT) compiler:

- ✓ The JIT Compiler **neutralizes** the **disadvantage** of the **interpreter.**
- ✓ The **Execution Engine** will be using the **help of the interpreter** in **converting byte code**, but **when it finds repeated code it uses the JIT compiler**, which compiles the entire **bytecode** and changes it to **native code**.
- ✓ The JIT Compiler compiles bytecode to machine code at runtime and **improves the**

JVM (Java Virtual Machine)

7.3 Garbage Collector:

- ✓ **Automatic freeing** of **Heap Memory**.
- ✓ **Collects** and **removes unreferenced objects**.
- ✓ Garbage Collection can be triggered by **calling System.gc().**

8) Java Native Interface (JNI):

- ✓ JNI will be **interacting** with the **Native Method Libraries** and **provides** the Native Libraries required for the **Execution Engine.**

Example:

- ✓ If we are running the Java application on Windows, then the **native method interface** will **connect** the **Windows libraries** (native method libraries) for **executing Windows methods** (native methods).

9) Native Method Libraries:

- ✓ This is a **collection of the Native Libraries**, which is **required for the Execution Engine**.

Java Byte Code (Java Magic)

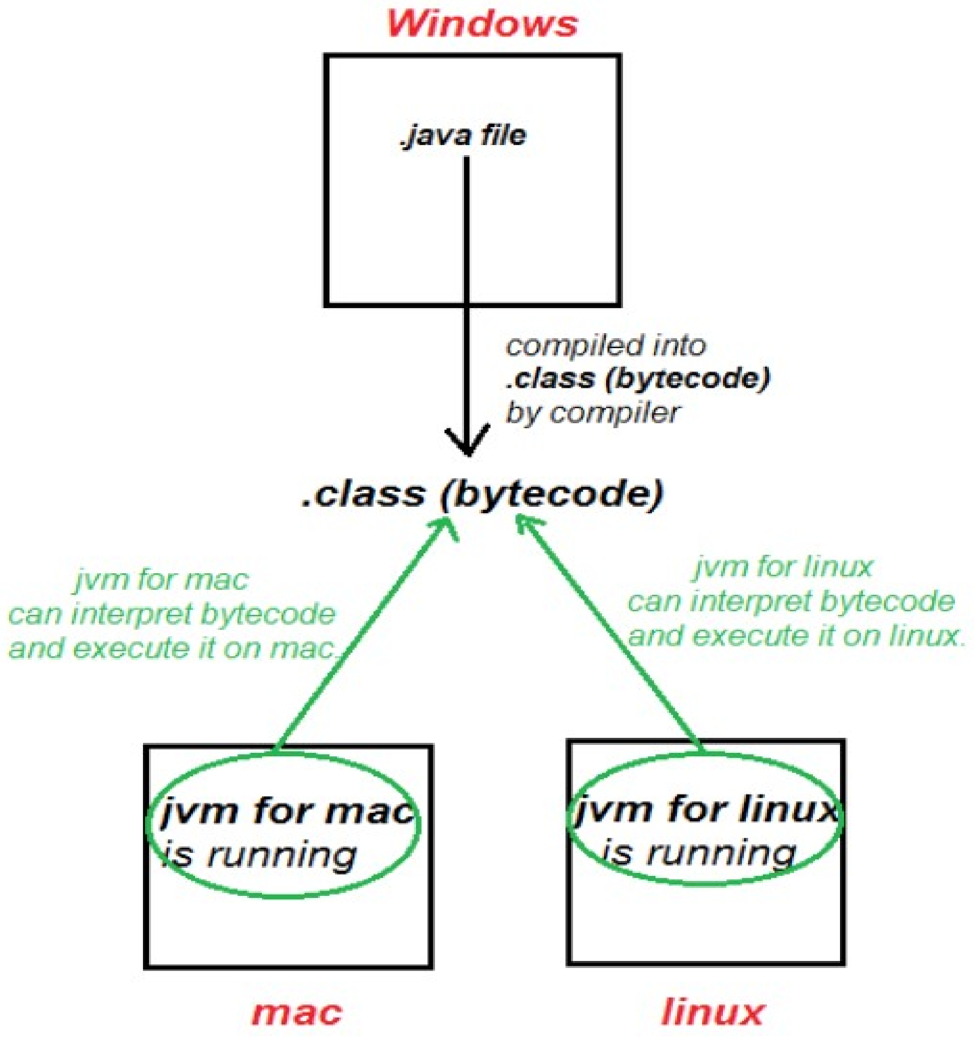
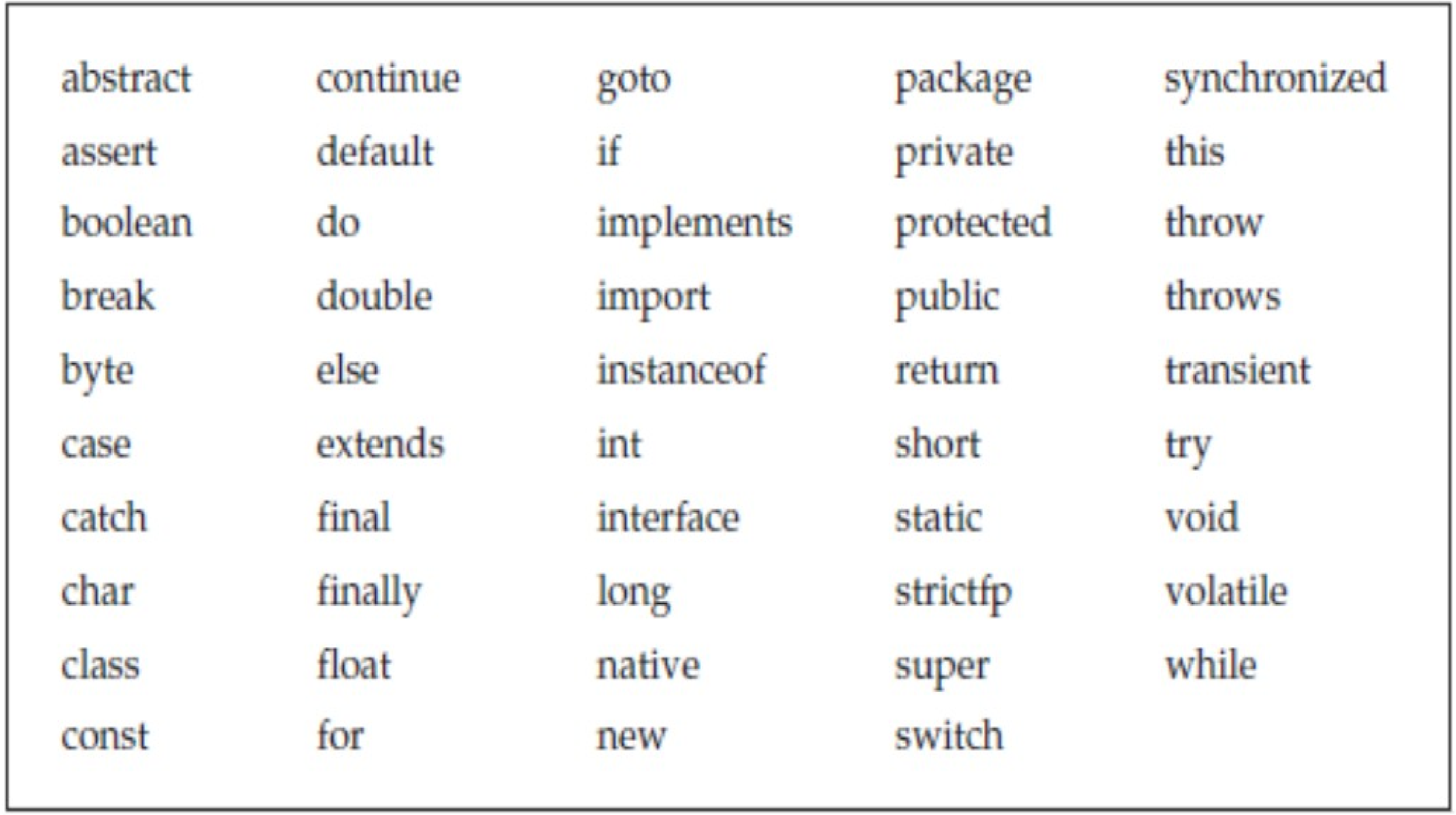


Fig. File was compiled on windows and could be executed on mac and linux.
Hence, making java platform independent.

JAVA KEYWORDS

- There are **49 reserved keywords** currently **defined** in the **Java** language
- These keywords **cannot be used** as **names for a variable, class**, or **method.**



abstract
assert
boolean
break
byte
case
catch
char
class
const
continue
default
do
double
else
extends
final
finally
float
for
goto
if
implements
import
instanceof
int
interface
long
native
new
package
private
protected
public
return
short
static
strictfp
super
switch
synchronized
this
throw
throws
transient
try
void
volatile
while

Data Types in Java

- "Data types **represent the different values** to be **stored in variable**"
- **Data type specifies:**
  - ✓ **Type of Value stored in a variable**
  - ✓ **Range of Values to a variable**
  - ✓ **Size of variable**
  - ✓ **Operations Performed**
- Java is a strongly typed language

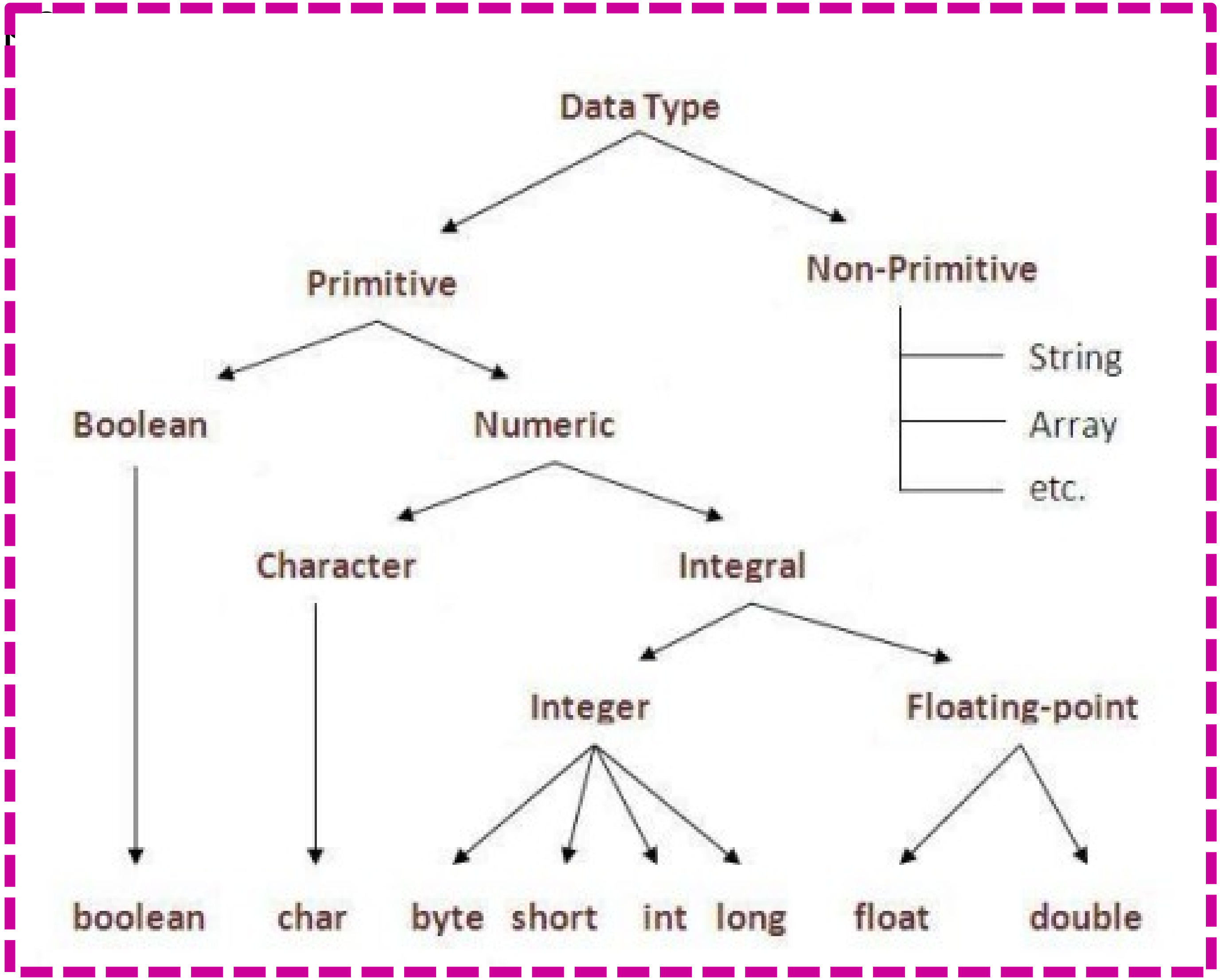
1. **Every variable has a type**, every expression has a type, and **every type is strictly defined.**
2. **All assignments,** whether explicit or via parameter passing in method calls, are **checked for Type compatibility.**
3. There are **no automatic conversions of conflicting types** as in **some languages.**
4. The **Java compiler checks all expressions** and **parameters** to ensure that the types

Data Types in Java

- In java, there are two types of data ty

  1. Primitive data types

  2. Non – Primitive Data Types

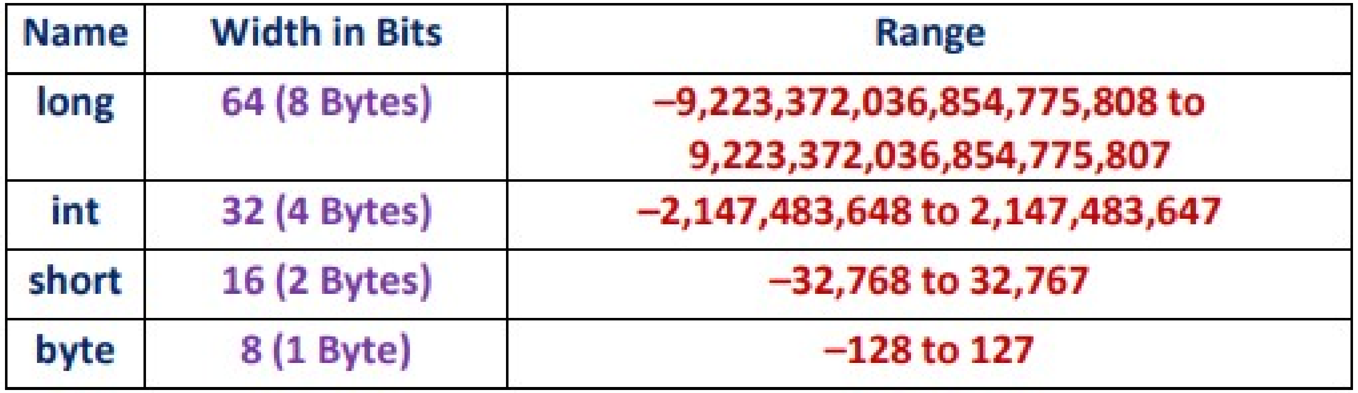


Data Types in Java

- All of **integer data types** are **signed**, **positive** and **negative values**.
- **Java does not support unsigned**, **positive-only integers**.
- Java implements the standard **IEEE–754** for **floating-point types.**
- There are **two kinds of floating-point types**, **float** and **double** which represent single precision (32bits) - and double precision (64bits) - numbers.
- **Java has** a **primitive type**, called **boolean**, for **logical values**. **It can have only** one of **two possible values**, **true or false.**

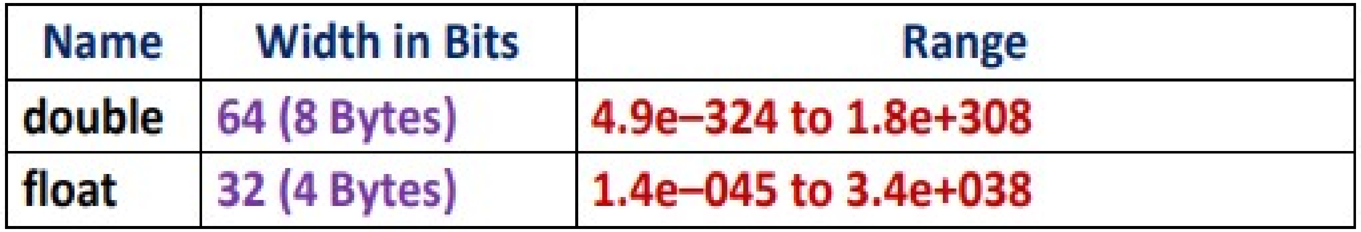
Data Types in Java

- Integer Type Range



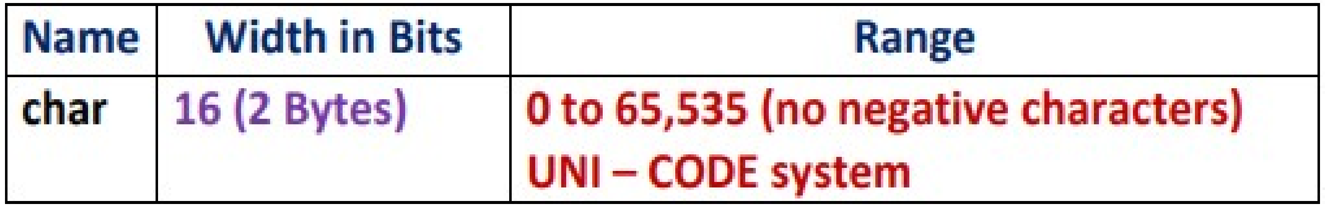
Name	Width in Bits	Range
long	64 (8 Bytes)	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32 (4 Bytes)	–2,147,483,648 to 2,147,483,647
short	16 (2 Bytes)	–32,768 to 32,767
byte	8 (1 Byte)	–128 to 127

- Floating Point Range



Name	Width in Bits	Range
double	64 (8 Bytes)	4.9e–324 to 1.8e+308
float	32 (4 Bytes)	1.4e–045 to 3.4e+038

- Character type Range



Name	Width in Bits	Range
char	16 (2 Bytes)	0 to 65,535 (no negative characters) UNI – CODE system

Java Character System

- **Java** uses **Unicode system**. Unicode is a **universal international standard character encoding** that is capable of **representing most of** the **world's** written **languages.**
- **Before Unicode**, there were **many** language **standards like ASCII** (American Standard Code for Information Interchange).
- In Unicode, **character holds 2 byte**, so **java also** uses **2 byte for characters.**
- Unicode defines a **fully international character set** that can **represent all of the characters** found in **all human languages** such as **Latin, Greek, Arabic, Cyrillic** etc.
- The **range** of a **char is 0 to 65,535**.
- There are **no negative chars.**
- Since **Java is designed** to allow programs **to be written for worldwide use**, it makes sense that **it would use Unicode** to **represent characters.**

Java Variables

- Variable is a **name of memory location**.
- A Variable Has
  - ✓ **NAME (VALID IDENTIFIER)**
  - ✓ **VALUE**
  - ✓ **TYPE**
  - ✓ **SCOPE & LIFETIME**

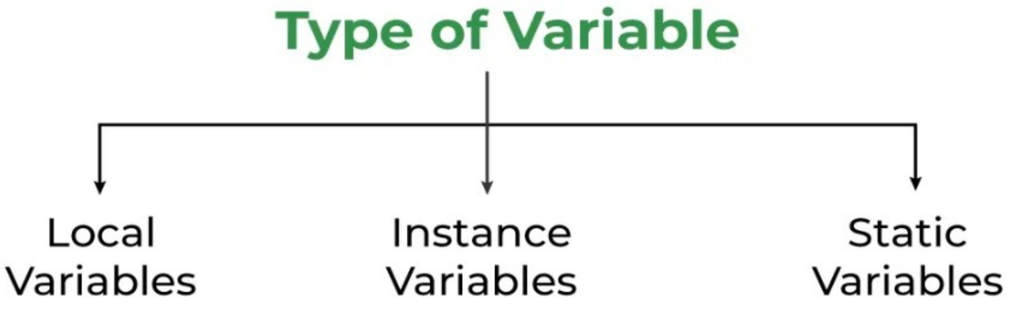
Syntax

```
datatype var_name = value;
```

```
int data=50; //Here data is variable
int x,y;
float p=3.142, result;
char grade ='A';
```

Types of Variable:

- There are three types of variables in java:



Type of Variable

Local Variables

Instance Variables

Static Variables

Java Variables

i. Local Variable:

- ✓ A **variable** which is **declared inside** the **method** is called **local variable.** (Stored in Stack Area of Memory)
- ✓ These variables are **created when** the **block is entered**, or the **function is called** and **destroyed after exiting from the block** or when the call returns from the function.
- ✓ The **scope** of these variables exists **only within the block.**

ii. Instance Variable:

- ✓ A **variable** which is **declared inside** the **class** but **outside** the **method**, is called **instance variable**. This is not declared as static. (Stored in Heap area of Memory).
- ✓ These variables are **created when an object** of the **class is created** and **destroyed** when the **object is destroyed.**

Java Variables

iii. Static variable:

- ✓ A **variable** that is **declared** as **static** is called **static variable**. It **cannot be local**. It **belongs to a class**. (Stored in Class Area of Memory).
- ✓ These static variables are declared **within a class outside of any method**, constructor, or block.
- ✓ we can only have **one copy of a static variable per class, irrespective of how many objects we create.**
- ✓ Static variables are **created at the start of program execution** and **destroyed automatically** when **execution ends.**

iv. Reference variable:

A **variable** that **refers to object** of a **class.** (Stored in Stack Area of Memory)

Java Variables

```
 Scanner sc = new Scanner(System.in);                   //sc-Reference
variable-stack
class A
{
     int data=50;                                     //instance variable-
heap
     static int m=100;                                  //static variable-
class area
     void display( )
     {
          int n=90;                              //local variable- stack
area
          -----------
          -----------
     }
 }
```

Java Variables

Example

```
public class Simple
{
       public static void main(String[]
args)
      {
      int a=10;
       int b=10;
       int c=a+b;
      System.out.println(“The sum is “
+c);
      }
}
```

Example

```
class DynInit
 {
      public static void main(String args[])
      {
             double a = 3.0, b = 4.0;
             a=a+5;
             b=b-1;
             double c = Math.sqrt(a * a + b * b);
            System.out.println("Result is " + c);
     }
 }
```

NOTE:

Java allows variables can be **initialized dynamically using any expression.**

Programs to Read Data from Standa Input

- Using **Java Scanner Class** to **read input.**
- Scanner is a class in **“java.util”** package used for obtaining the input of the primitive types like int, double, and strings etc.
- It is the **easiest way to read input** in a Java program.
- The **Java Scanner class breaks** the **input into tokens** using a **delimiter** that is **whitespace** by default.
- **Commonly used methods** of Scanner class

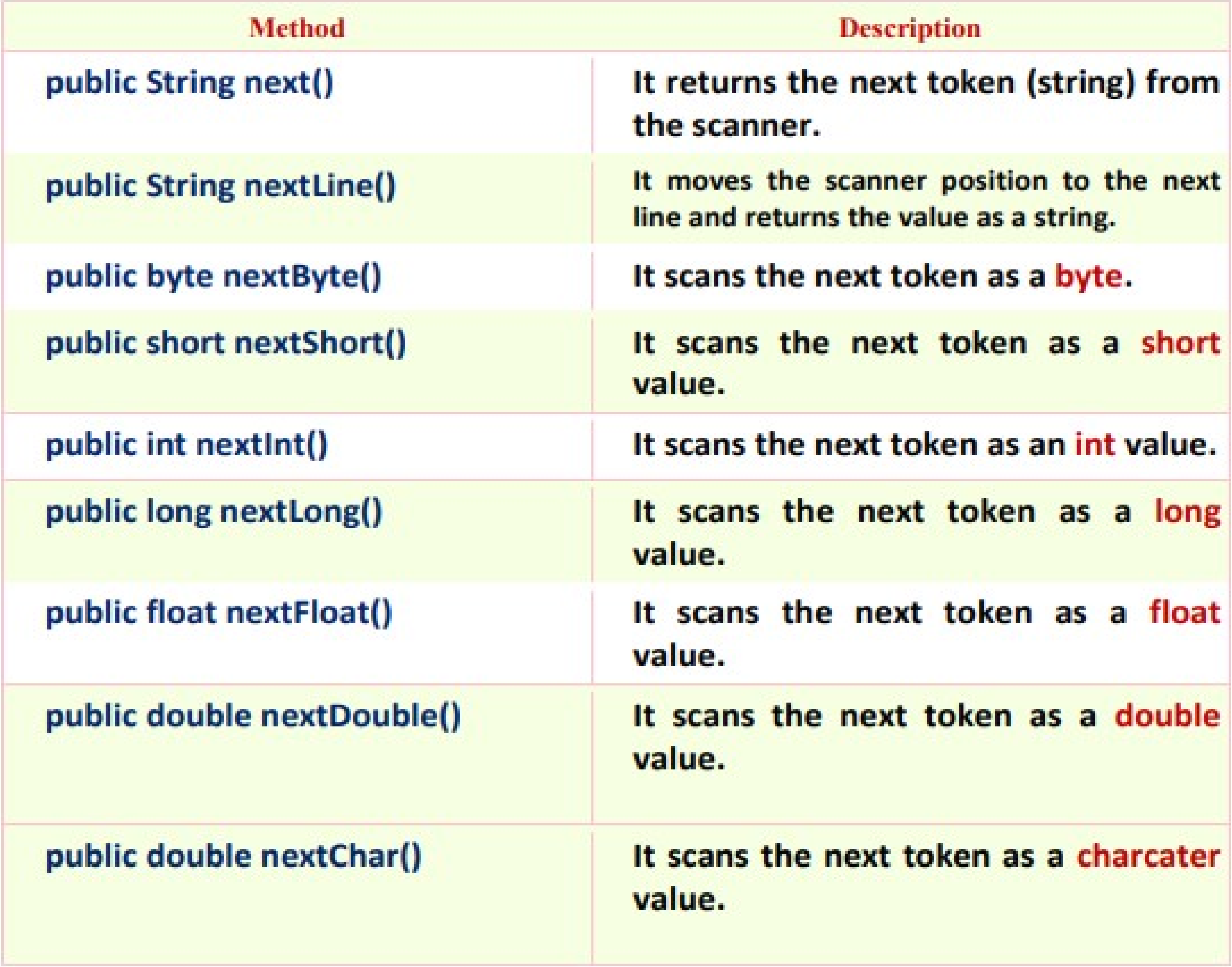
Syntax

```
Scanner sc=new Scanner(System.in);

datatype Variable_Name=sc.method_name( );
```

Example

```
Scanner sc=new Scanner(System.in);

int rollno=sc.nextInt( );
```

Programs to Read Data from Stand
Input



Method	Description
public String next()	It returns the next token (string) from the scanner.
public String nextLine()	It moves the scanner position to the next line and returns the value as a string.
public byte nextByte()	It scans the next token as a byte.
public short nextShort()	It scans the next token as a short value.
public int nextInt()	It scans the next token as an int value.
public long nextLong()	It scans the next token as a long value.
public float nextFloat()	It scans the next token as a float value.
public double nextDouble()	It scans the next token as a double value.
public double nextChar()	It scans the next token as a charcater value.

// Example to read and display data

```
import java.util.*;
import java.io.*;
public class DataRead
{
    public static void main(String args[])
    {
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter your rollno");
        int rollno=sc.nextInt();
        System.out.println("Enter your name");
        String name=sc.next();
        System.out.println("Enter your fee");
        double fee=sc.nextDouble();
        System.out.println(“Roll No : “ +rollno);
        System.out.println(“Name is “ +name);
        System.out.println(“Feee is “ +fee);
        //System.out.println("Rollno:"+rollno +" name:" +name + " fee:" +fee);
        sc.close();
    }
}
```

```
//Program to read a number and check it is even or odd.
import java.util.*;
public class Even
{
    public static void main(String[ ] args)
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Entera a number");
        int n = sc.nextInt();
        if( n % 2 == 0)
            System.out.println("Even Number " +n);
        else
            System.out.println("Odd Number "+n);

    }
}
```

```
//Program to find the average of a student in three subjects marks (Double).
import java.util.*;
public class Grade
{
	public static void main(String[] args)
	{
		Scanner sc = new Scanner(System.in);
		System.out.println("Entera the Marks in 3 Subjects");
		double m1 = sc.nextDouble();
		double m2 = sc.nextDouble();
		double m3 = sc.nextDouble();
		double avg = (m1+m2+m3)/3;
		System.out.println("The average is "+avg);
	}
}
```

Type Conversions and Type Casting

- When you **assign value of one data type to another**, the **two types might not be compatible** with **each other.**
- If the **data types are compatible**, then **Java will perform** the **conversion automatically** known as **Automatic Type Conversion.**

  Example: it is always **possible to assign an int value** to **a long variable.**
- If the **data types are not compatible** then **they need to be casted** or **converted explicitly.**

  Example: there is **no automatic conversion defined** from **double** to **byte.**

Java is a strongly typed language

- ✓ **Every variable has a type**, every **expression has a type**, and **every type is strictly defined.**
- ✓ All assignments, whether explicit or via parameter passing in method calls, are checked for Type compatibility.
- ✓ There are **no automatic conversions** of **conflicting types** as in **some languages.**

Type Conversions and Type Casting

Java Type conversion are in **two forms :**

1. Automatic Type Conversion / Widening / Type Coersion

- When **one type of data is assigned** to **another type of variable**, an automatic type conversion will take place if the following two conditions are met:
  1. The **two types are compatible.**
  2. The **destination** type is **larger than** the **source type.**
- When these two conditions are met, a widening conversion takes place.

Example: **int type** is **always** large **enough to hold all valid byte values**, so **no explicit cast statement** is **required**.

For widening conversions

- The **numeric types**, including **integer and floating-point types**, are **compatible** with **each other.**

Eg: int a=10;

float f=a;

Type Conversions and Type Casting

- **No automatic conversions** from the **numeric types** to **char** or
- **char** and **boolean** are **not compatible** with **each other**. (In compatible Types)

2. Explicit Type Conversion / Narrowing / Type Casting

- The **automatic type conversions** will **not fulfill** all **needs.**
- For example an **int value** to a **byte variable conversion will not** be **performed automatically**, because a byte is smaller than an int.

```
int m = 67;
byte b = m;
System.out.println("The value of m is " +m); //67
System.out.println("The value of ch is " +b);
```

- **This kind of conversion** is **sometimes** called a **narrowing conversion**, since we are **explicitly making** the **value narrower so that it will fit** into the **target type.**
- To create a **conversion between two incompatible** types, we **must use a cast.**

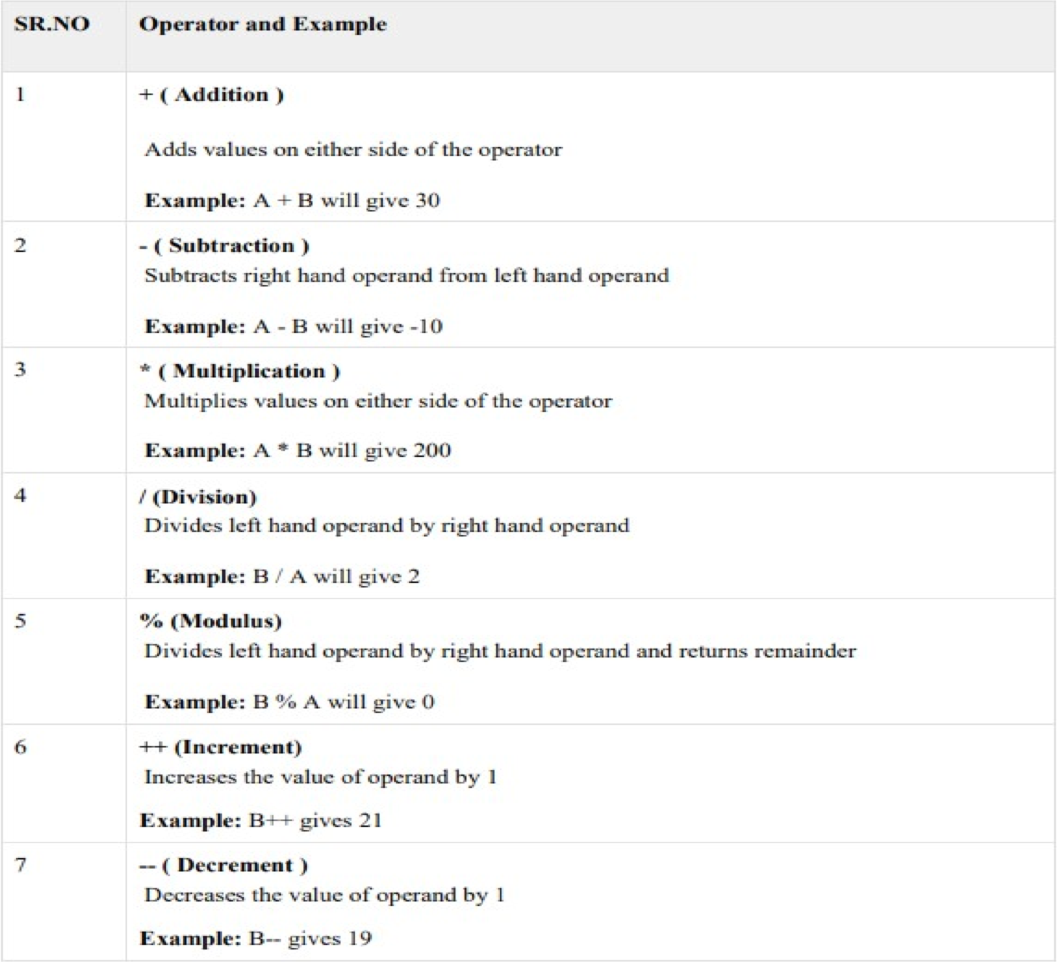
Operators in Java

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

- **Arithmetic Operators**
- **Relational Operators**
- **Bitwise Operators**
- **Logical Operators**
- **Assignment Operators**
- **Conditional Operators**

Operators in Java

i. Arithmetic Operators:



SR.NO	Operator and Example
1	**+ (Addition)** Adds values on either side of the operator **Example:** A + B will give 30
2	**- (Subtraction)** Subtracts right hand operand from left hand operand **Example:** A - B will give -10
3	*** (Multiplication)** Multiplies values on either side of the operator **Example:** A * B will give 200
4	**/ (Division)** Divides left hand operand by right hand operand **Example:** B / A will give 2
5	**% (Modulus)** Divides left hand operand by right hand operand and returns remainder **Example:** B % A will give 0
6	**++ (Increment)** Increases the value of operand by 1 **Example:** B++ gives 21
7	**-- (Decrement)** Decreases the value of operand by 1 **Example:** B-- gives 19

```
class OperatorDemo
 {
      public static void main(String args[])
      {
            int a=10,b=5;
            System.out.println(a+b); //15
            System.out.println(a-b); //5
            System.out.println(a*b); //50
            System.out.println(a/b); //2
             System.ot.println(a%3); //1
            double c=42.25;
           System.out.println(c%10);
//2.25possiblein java
      }
}
```

Operators in Java

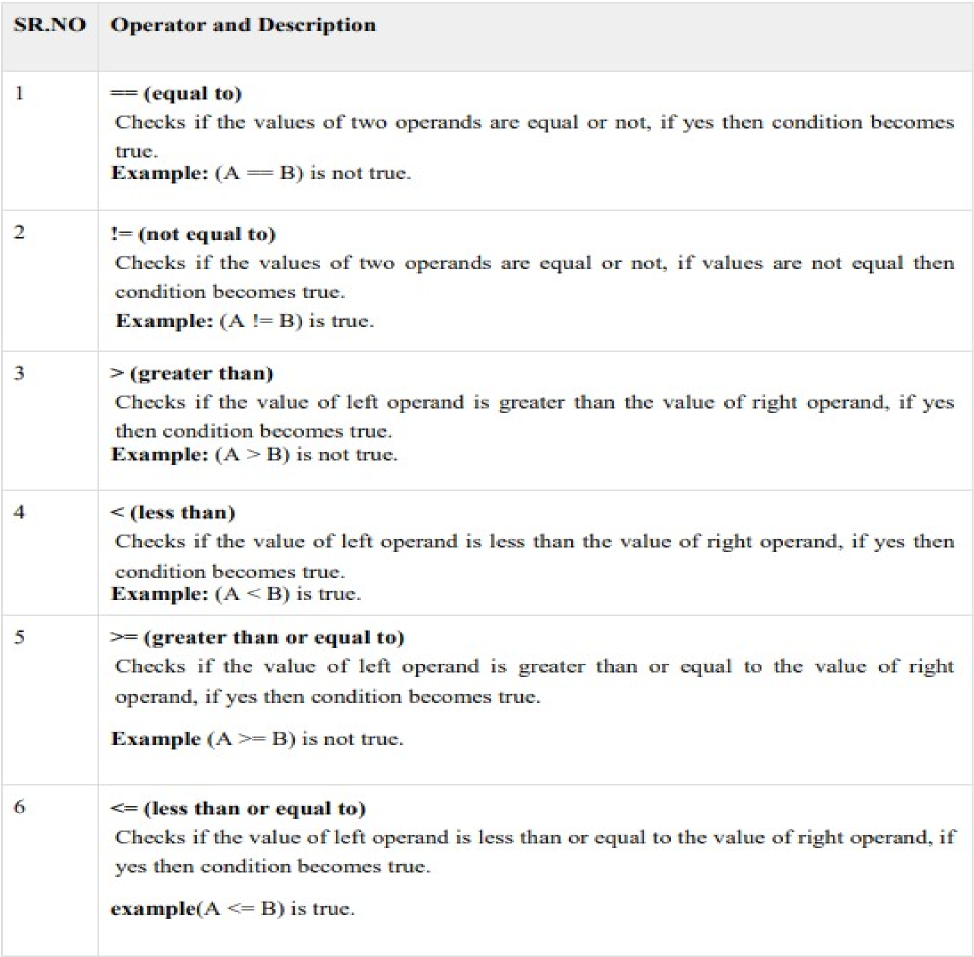
i. Arithmetic Operators:

```
class OperatorDemo
 {
      public static void main(String 
args[])
     {
          int x=10;
         System.out.println(x++); //10 
(11)
         System.out.println(++x); //12
         System.out.println(x--); //12 
(11)
         System.out.println(--x); //10
     }
}
```

```
class OperatorDemo
 {
       public static void main(String args[])
      {
             int a=10;
             int b=10;
             System.out.println(a++ + +
+a);//10+12=22
             System.out.println(a); //12
             System.out.println(b++ + b+
+);//10+11=21
             System.out.println(b); //12
      }
}
```

Operators in Java

ii. Relational Operators (Comparing two things):

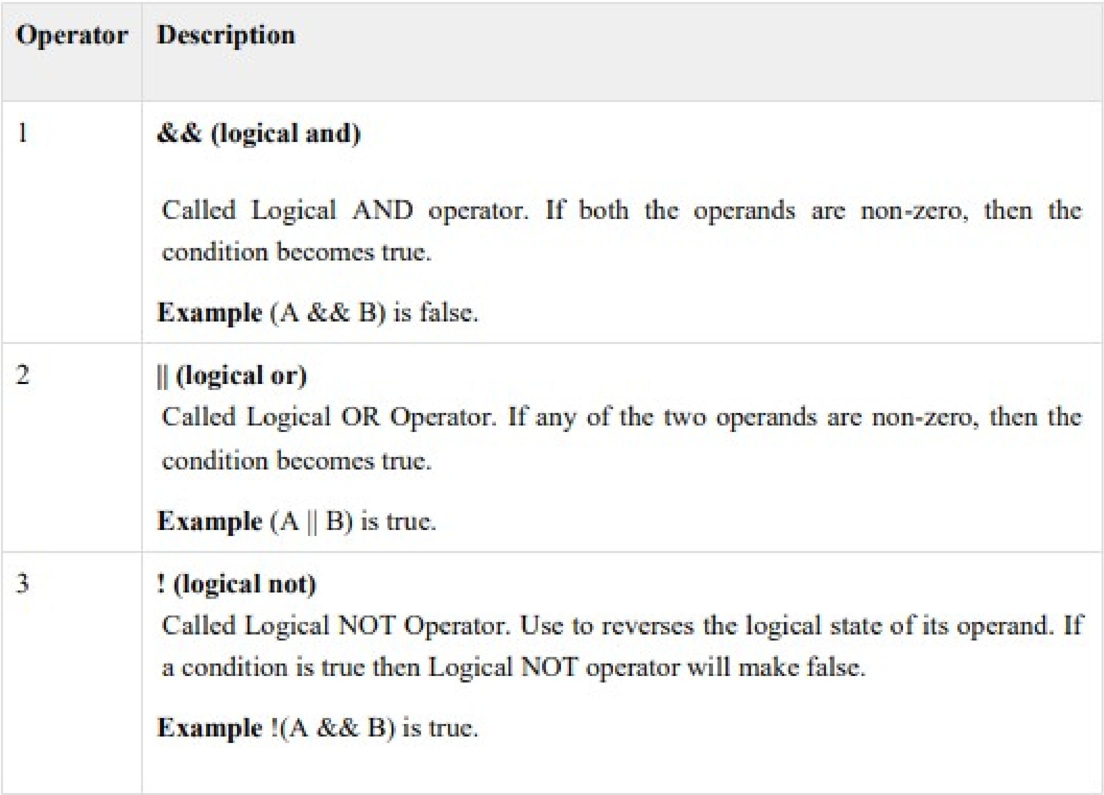


SR.NO	Operator and Description
1	**== (equal to)** Checks if the values of two operands are equal or not, if yes then condition becomes true. **Example:** (A == B) is not true.
2	**!= (not equal to)** Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. **Example:** (A != B) is true.
3	**> (greater than)** Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. **Example:** (A > B) is not true.
4	**< (less than)** Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. **Example:** (A < B) is true.
5	**>= (greater than or equal to)** Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. **Example** (A >= B) is not true.
6	**<= (less than or equal to)** Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. **example**(A <= B) is true.

```
class OperatorDemo
 {
      public static void main(String args[])
     {
          int a = 4;
          int b = 1;
          boolean c = a < b;
          System.out.println("The C value is "
+ c) //false
          System.out.println(a==4) //True
          System.out.println(a!=b) //False
          System.out.println(a<=4) //True
       }
}
```

Operators in Java

iii. Logical Operators (Combining two or more relations):

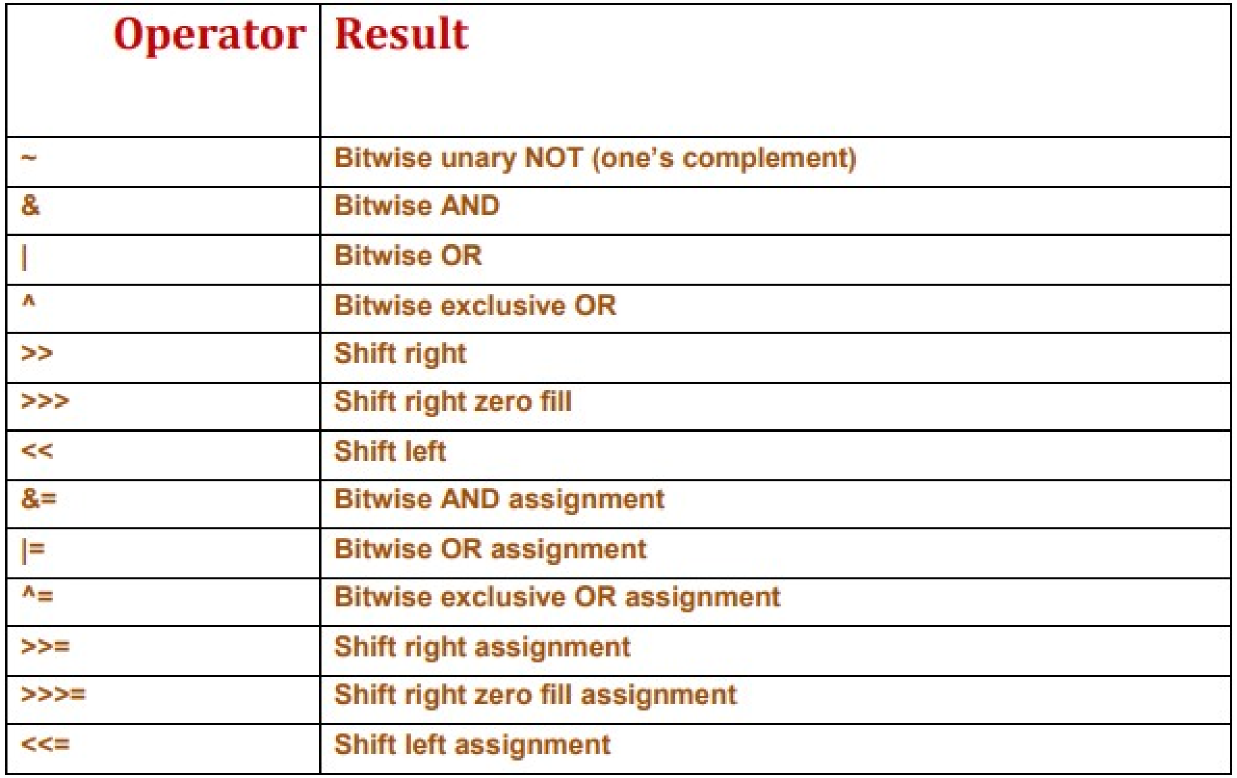


Operator	Description
1	**&& (logical and)** Called Logical AND operator. If both the operands are non-zero, then the condition becomes true. **Example** (A && B) is false.
2	**|| (logical or)** Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true. **Example** (A || B) is true.
3	**! (logical not)** Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false. **Example** !(A && B) is true.

```
class OperDemo
 {
          public static void main(String 
args[])
          {
                 int a=10;
                 int b=5;
                 int c=20;

System.out.println(a<b&&a<c);
                 System.out.println(a<b||
a<c);
             }
 }
```

Operators in Java

iv. Bitwise Operators (Operates at bit level – on binary data):



Operator	Result
~	Bitwise unary NOT (one's complement)
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

```
class Operator
 {
      public static void main(String
args[])
    {
         byte a=10;
         byte b=8;
         byte c;
        c = a | b;
        System.out.println(c);
        c = a & b;
        System.out.println(c);
       c = a ^ b;
       System.out.println(c);
    }
 }
```

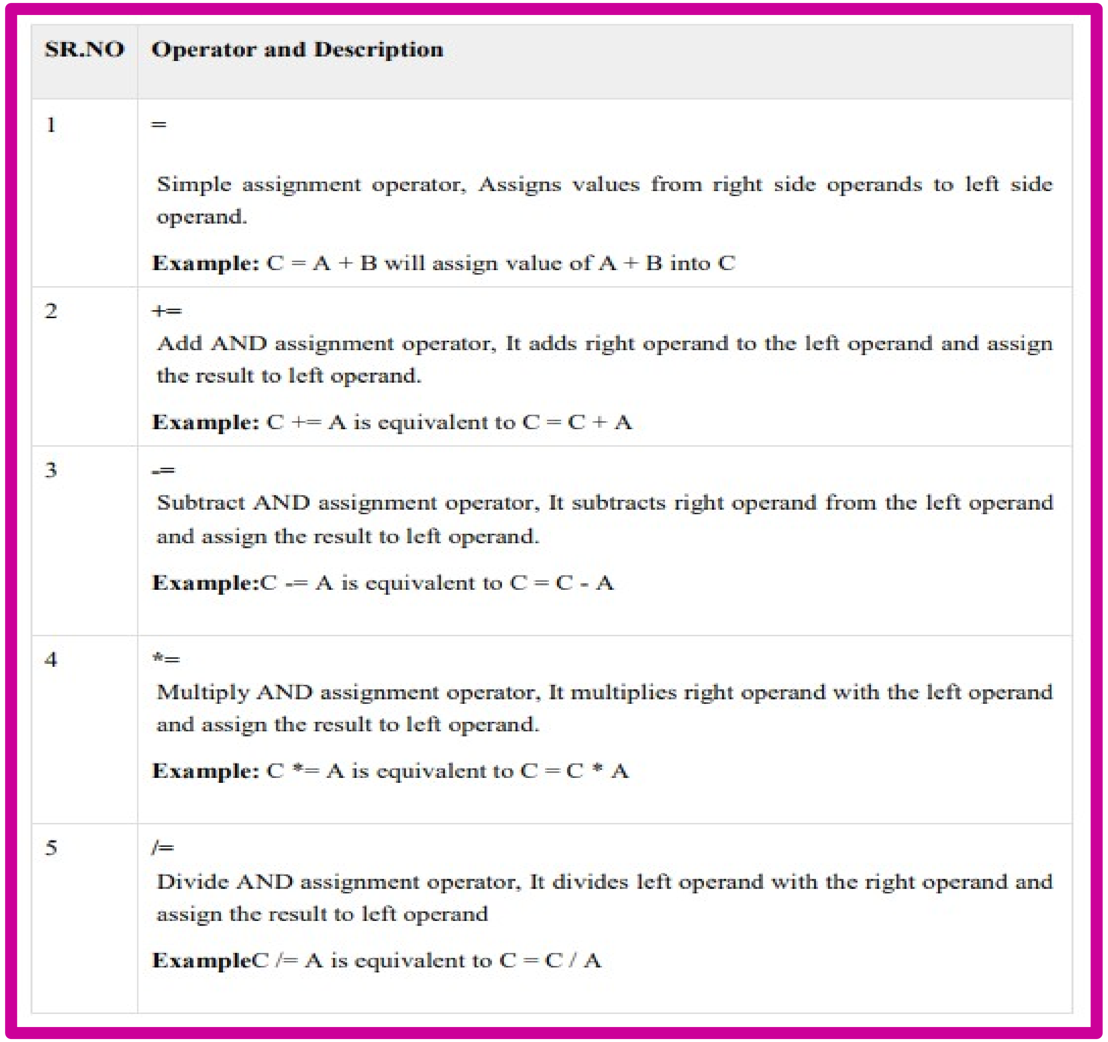
Operators in Java

iv. Bitwise Operators:

```
class Oper
 {
          public static void main(String args[])
         {
               byte a=10;
               System.out.println(a<<2); //40
               byte b=-15;
              System.out.println(b<<3); //-120
               byte c=10;
               System.out.println(c>>2); //2
               byte d=-15;
               System.out.println(d>>1); //-8
         }
}
```

Operators in Java

v. Assignment Operators:



SR.NO	Operator and Description
1	= Simple assignment operator, Assigns values from right side operands to left side operand. **Example:** C = A + B will assign value of A + B into C
2	+= Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand. **Example:** C += A is equivalent to C = C + A
3	-= Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand. **Example:**C -= A is equivalent to C = C - A
4	*= Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand. **Example:** C *= A is equivalent to C = C * A
5	/= Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand **Example**C /= A is equivalent to C = C / A

Operators in Java

v. Conditional Operator:

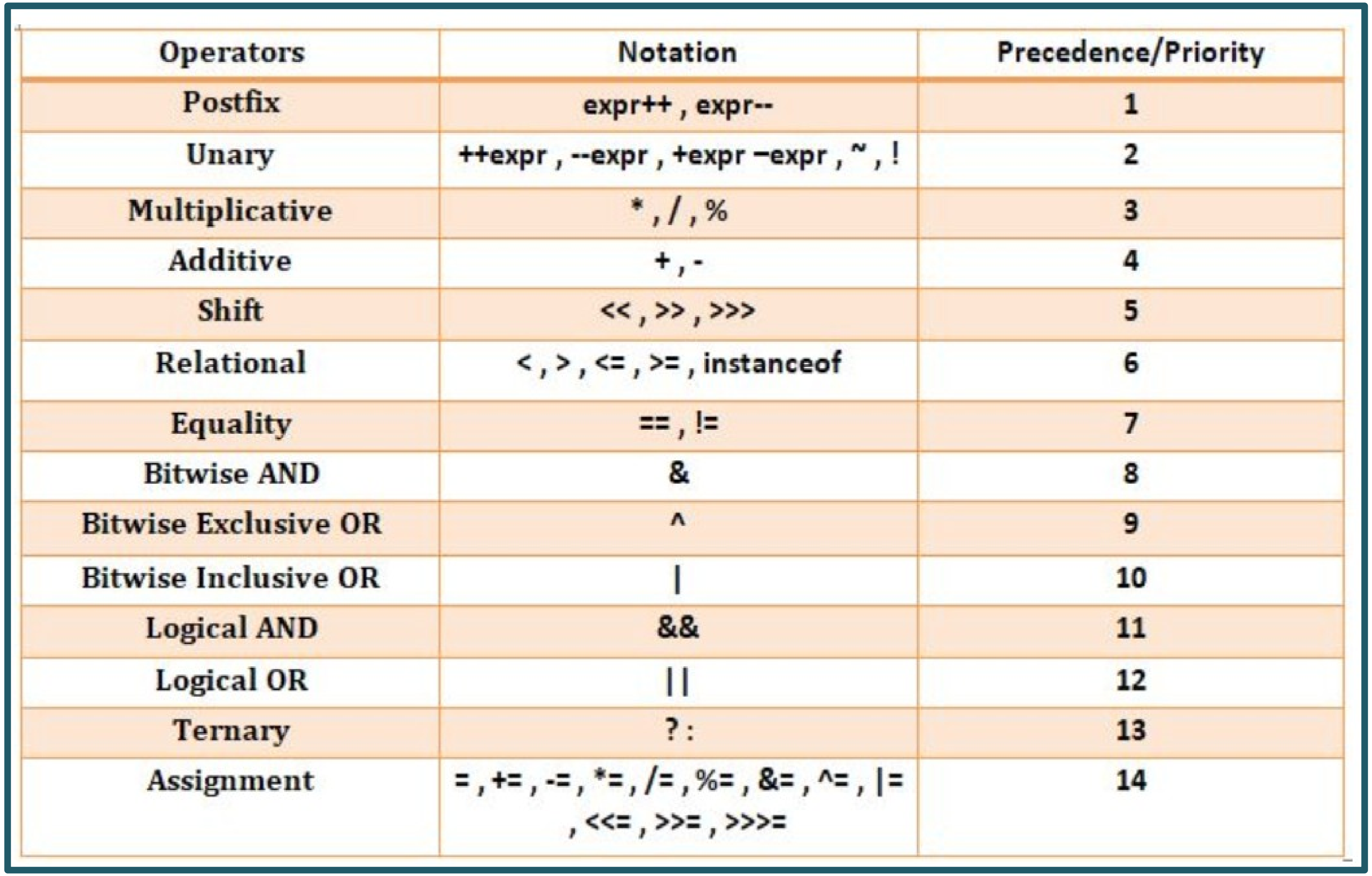
- Conditional operator is also known as the **ternary operator.**
- This operator consists of **three operands** and is used to evaluate expressions.
- The **goal of the operator is to decide which value should be assigned to the variable.**

Syntax

variable x = (expression) ? value if true : value if false

```
class Test
{
        public static void main(String args[ ])
       {
        int a, b;
        a = 10;
        b = (a == 1) ? 20: 30;
        System.out.println( "Value of b is : " +
       b );
        b = (a == 10) ? 20: 30;
        System.out.println( "Value of b is : " +
       b );
        }
       }
```

Operator Precedence Table in Java



Operators	Notation	Precedence/Priority
Postfix	expr++ , expr--	1
Unary	++expr , --expr , +expr −expr , ~ , !	2
Multiplicative	* , / , %	3
Additive	+ , -	4
Shift	<< , >> , >>>	5
Relational	< , > , <= , >= , instanceof	6
Equality	== , !=	7
Bitwise AND	&	8
Bitwise Exclusive OR	^	9
Bitwise Inclusive OR	|	10
Logical AND	&&	11
Logical OR	||	12
Ternary	? :	13
Assignment	= , += , -= , *= , /= , %= , &= , ^= , |= , <<= , >>= , >>>=	14

Control Statements

1. Conditional / Selection Statements:

-There are **various types** of **selection stateme**

1. if statement
2. if-else statement
3. Nested if statement
4. if-else-if ladder
5. Switch Statement

i. if statement

```
if(condition)
 {
     //code to be executed
 }
```

ii. if-else statement

```
if(condition)
 {
        //code to be executed
 }
else
{
       //code to be executed
}
```

iii. Nested if statement

```
if(condition1)
 {
        if(condition2)
        {
            //code to be executed
        }
        //code to be executed
 }
```

Control Statements

iv. if-else-if ladder

```
if(condition1)
 {
      //code to be executed
 }
else if(condition2)
{
      //code to be executed
}
else
{
        //code to be executed
}
switch(expression)
 {
     case value1: //code to be
executed;
                    break;
     case value2: //code to be
executed;
                                        break;
                    ................................
..........
                    default: //code to be
executed;
```

```
class Example
{
      public static void main(String[] args)
      {
            int number=20;
            switch(number)
            {
             case 10: System.out.println("CSE");break;
             case 20: System.out.println("EEE");break;
             case 30: System.out.println("ECE");break;
            default:System.out.println("Invalid input");
            }
      }
}
```

Switch Statement is fall-through

- The java switch statement is fall-through. It **means it executes all statement after first match** if **break statement** is **not used** with switch cases.

```
class Example2
 {
     public static void main(String[] args)
     {
          int number=20;
          switch(number)
          {

               case 10: System.out.println("10");

               case 20: System.out.println("20");

               case 30: System.out.println("30");

               default:System.out.println("Hello");
          }
     }
 }
```

Control Statements

2. Loop / Repetitive / Iterative Statements :

i. while Loop:

- ✓ The Java while loop is used to **iterate a part of the program several times.**
- ✓ If the **number of iteration is not fixed**, it is **recommended to use** while loop.
- ✓ It is an **entry controlled loop.**

Syntax:

```
while(condition)
 {
       //code to be
executed
 }
```

Java Infinitive While Loop

- ✓ If you **pass true** in the while loop, it will be **infinitive while loop.**

```
while(true)
 {
       System.out.println("never
ends");
 }
```

Control Statements

ii.do-while Loop:

- ✓ do-while loop is used to iterate a part of the program several times.
- ✓ If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.
- ✓ The Java do-while loop is executed at least once because condition is checked after loop body.
- ✓ It is an exit controlled loop

Syntax

```
do
{
        //code to be
executed
 }while(condition);
```

- ✓ If you pass true in the do-while loop, it will be infinitive do-while loop.

```
do
{

System.out.println("Never
Ends");
}while(true);
```

Control Statements

iii. for Loop:

- ✓ The Java **for loop is used to iterate** a part of the program **several times.** If the **number of iteration is fixed**, it is **recommended** to **use for loop.**
- ✓ It is an **entry controlled loop.**
- ✓ There are two types of for loop in java.
  - **a. Simple For Loop**
  - **b. for-each or Enhanced For Loop**

a. Simple for Loop:

- ✓ The **simple for loop is same as C/C++.** We can initialize variable, check condition an increment/decrement value.

Syntax

```
for(initialization;condition;incr/
                decr)
{
        //code to be executed
}
```

Example

```
for(int i=1;i<=10;i++)
 {
        System.out.println(i);
 }
```

Control Statements

b. Java for-each Loop:

✓ The **for-each loop** is **used to traverse array** or **collection in java.** It is **easier** to use than **simple for loop** because we **don't need to increment** value and use subscript notation.

✓ It works on **elements basis not index.**

✓ It **returns element one by one** in the defined variable.

Syntax

```
for(type var: array)
{
        //code to be
executed
}
```

Example

```
int list[] = { 11, 2, 3, 14, 5, 62, 7,
8, 9, 10 };
int sum = 0;
 for(int x : list)
 {
          System.out.println("Value
is: " + x);
          sum = sum + x;
 }
```

```
Java Infinitive for Loop
for( ; ; )
 {
          System.out.println("Never
Ends");
 }
```

Java break Statement

- The Java break is used to **break loop** or **switch statement**. It breaks the current flow of the program at specified condition.
- When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.
- **In case of inner loop**, it **breaks only inner loop.**

Example

```
for(int i=1;i<=10;i++)
 {
       if(i==5)
      {
           break;
      }
      System.out.println(i);
}
```

Java Continue Statement

- The Java **continue statement** is used to **continue loop.**
- We might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration.
- It **continues the current flow** of the program and **skips the remaining code at specified condition.**
- In case of inner loop, it continues only inner loop.

Example

```
for(int i=1;i<=6;i++)
 {
      if(i==3)
      {
            continue;
      }
      System.out.println(i);
}
```

Java return Statement

- The return statement is used to **explicitly return from a method**. That is, it **causes program control to transfer back to the caller** of the method.
- Thus, the return statement **immediately terminates the method in which it is executed.**

```
class Demo
{
	public static void main(String[] args)
	{
		int res;
		res = add(20,30);
		System.out.println("The result is "+res);
	}
	public static int add(int a , int b)
	{
	return(a+b);
	}
}
```

Arrays In Java

An array is **a group of like-typed (homogeneous) elements** that are **referred** to **by** a **common name.**

- ✓ **Arrays of any type** can be **created** and **may have one or more dimensions.** (One ,Two or More)
- ✓ A **specific element** in an **array is accessed** by **its index.** (index starts from 0)
- ✓ The **memory for an array is dynamically allocated** unlike C or C++.

i. One-Dimensional Arrays:

- ✓ A one y, a **list of**
- ✓ The **g** al array d

Syntax:

Datatype var-name[ ];

Example

int days[ ];
double avg[ ];

- ✓ **Although this declaration** an array variable, **no array actually exists.**

Arrays In Java

- ✓ The **array is set to null**, which represents an **array with no value.**
- ✓ To create **physical array of integers**, we must **allocate using new operator.**
- ✓ **new** is a **special operator** that **allocates memory dynamically.**
- ✓ A one-dimensional array can be created as

Syntax:

array-var = new type [size]; //creation or allocation

Example

int days = new int[7];
double avg = new double[60];

- ✓ Both **declaration** and **allocation can be combined as**

Syntax:

Datatype array-var = new Datatype [size]; //creation or allocation

Example

int days = new int[7];
double avg = new double[60];

Arrays In Java

```
class Array
{
        public static void main(String
args[])
       {
             int days[] = new int[7];
            days[0] = 12; //assigning a
value at index
            days[1] = 22;
            days[2] = 92;
            days[3] = 28;
            days[4] = 32;
            days[5] = 40;
            days[6] = 39;
           //Accessing an array
           for(int i=0; i<6;i++)
           {
               System.out.println( days[i]);
           }
        }
}
```

Array Initialization

- ✓ Arrays **can be initialized when they are declared.**
- ✓ An array initializer is a **list of comma-separated** expressions **Surrounded by curly braces**.
- ✓ The commas separate the values of the array elements.

Syntax:
Datatype arr_name[size] = { List of elements separated by comma};

Example
int days[7] = {7,9,5,12,45,23,10};

- ✓ The **size of array need not to be specified.**

```
class Average
{
       public static void main(String args[])
       {
          double a[] = {75.1, 76.2, 65.3, 77.4, 
84.5};
          double sum = 0;
           int i;
           for(i=0; i<5; i++)
                 sum = sum + a[i];
           System.out.println("Average is " + 
sum / 5);
       }
}
```

Multi-Dimensional Arrays

- In Java, multidimensional **arrays are actually arrays of arrays.**
- To declare a multidimensional array variable, **specify each additional index using another set of square brackets**.

Example:

A two-dimensional array can be declared and allocated as

```
int two[ ][ ] = new int[4][5];
```

- **When you allocate memory** for a multidimensional array we **need to** only **specify** the memory for the **first (leftmost) dimension.**

- We can **allocate the remaining dimensions separately.**

```
int a[][] = new int[4][ ];
a[0] = new int[3];
a[1] = new int[3;
a[2] = new int[3];
a[3] = new int[3];
```

```
class TwoD
{
        public static void main(String
args[])
        {
                int a[][]= new int[4][5];
                int i, j, k = 0;
                for(i=0; i<4; i++)
                for(j=0; j<5; j++)
                {
                      a[i][j] = k;
                       k++;
                }
                 for(i=0; i<4; i++)
                {
                      for(j=0; j<5; j++)

System.out.print(a[i][j] + " ");
                      System.out.println();
                  }
            }
}
```

Multi-Dimensional Arrays

- We can allocate **each row with different number of elements**; such an array is called **Jagged / Ragged Array.** i.e. Each row can be of different columns.

```
int a[][] = new int[4][ ];
a[0] = new int[2];
a[1] = new int[3];
a[2] = new int[4];
a[3] = new int[5];
```

- A multi dimensional array can be initialized as

```
int a[3][3] = {{1,2,3}, {3,4,5},{6,7,8}};
 int a[3][] = {{1,2,3}, {3,4,5},{6,7,8}};
int a[4][] = {{1,2,3,4}, {3,4 },{6,7,8,9},{1,2}}; //Jagged array
```

Multi-Dimensional Arrays

```
import java.io.*;
import java.util.*;
//Jagged or Ragged Array
class ArrDemo
{
       public static void main(String[] args)
       {
            Scanner sc = new Scanner(System.in);
            System.out.println("Enter the number of rows");
            int m = sc.nextInt();
             int a[][] = new int[m][ ];
             int n;
                      for(int i =0; i< m;i++)
                     {
                     System.out.println("Enter the number of
elements in " +i);
                     n = sc.nextInt();
                     a[i] = new int[n];
                    }
```

```
  System.out.println("Enter the
elements");
  for(int i = 0; i<m ; i++)
 for(int j=0;j<a[i].length;j++)
 a[i][j] = sc.nextInt();
 System.out.println("the matrix
elements are");
 for(int i = 0; i<m ; i++)
 {
 for(int j=0;j<a[i].length;j++)
 System.out.print(" " +a[i][j]);
 System.out.println();
}
 }
}
```

```
//3. Matrix Multiplication
import java.util.*;
public class MatrixDemo
{
public static void main(String args[])
{
 Scanner sc = new Scanner(System.in);
 System.out.println("Enter the size of Matrix A - m * n");
 int m = sc.nextInt();
 int n = sc.nextInt();

 System.out.println("Enter the size of Matrix B - p * q");
 int p = sc.nextInt();
 int q = sc.nextInt();
 if (n != p)
 {
 System.out.println("Multiplication cannot be performed");
 System.exit(0);
 }
 int a[][] = new int[m][n];
 int b[][] = new int[p][q];
 int c[][] = new int[m][q];


 System.out.println("Enter the elements of
Matrix-A");
 for(int i =0; i<m;i++)
 for(int j=0;j<n;j++)
 a[i][j] = sc.nextInt();
 System.out.println("Enter the elements of
Matrix-B");
 for(int i =0; i<p;i++)
 for(int j=0;j<q;j++)
 b[i][j] = sc.nextInt();
 // Actual Code
for(int i =0 ; i<m ;i++)
 {
 for(int j=0;j<q;j++)
 {
 c[i][j]=0;
 for(int k=0;k<p;k++)
c[i][j] = c[i][j] + a[i][k]*b[k][j];
}
}
```

```
 System.out.println("The Product Matrix-C is");
for(int i =0; i<m;i++)
 {
for(int j=0;j<q;j++)
 System.out.print(" " + c[i][j]);
System.out.println();
 }
}
}
```

- **Alternative Array Declaration Syntax**

  type[ ] var-name;
  int a[] = new int[3];
  int[] a1 = new int[3];
  char a[][] = new char[3][4];
  char[][] a = new char[3][4];
- when **declaring several arrays at the same time.**

  int[] a,b,c; // create three arrays
  char[][] a,b,c;

Classes and Objects

- The class is the **logical construct** upon which the entire **Java language is built .**
- The **class** forms the **basis** for **object-oriented programming** in Java.
- **Any concept** we implement **in a Java** program **must be encapsulated** within a **class.**
- A class is **declared by use** of the **class keyword**.
- A **class defines a new data type.**
- A class is a **template for an object**, and an **object** is an **instance** of a **class.**
- A **class contains data** and the **code** that operates on that data.
- The **data, or variables**, defined **within** a **class** are called **instance variables**, because **each object** of the **class contains** its **own copy** of these **variables.**
- The **methods** and **variables** defined **within a class** are called **members of the class.**

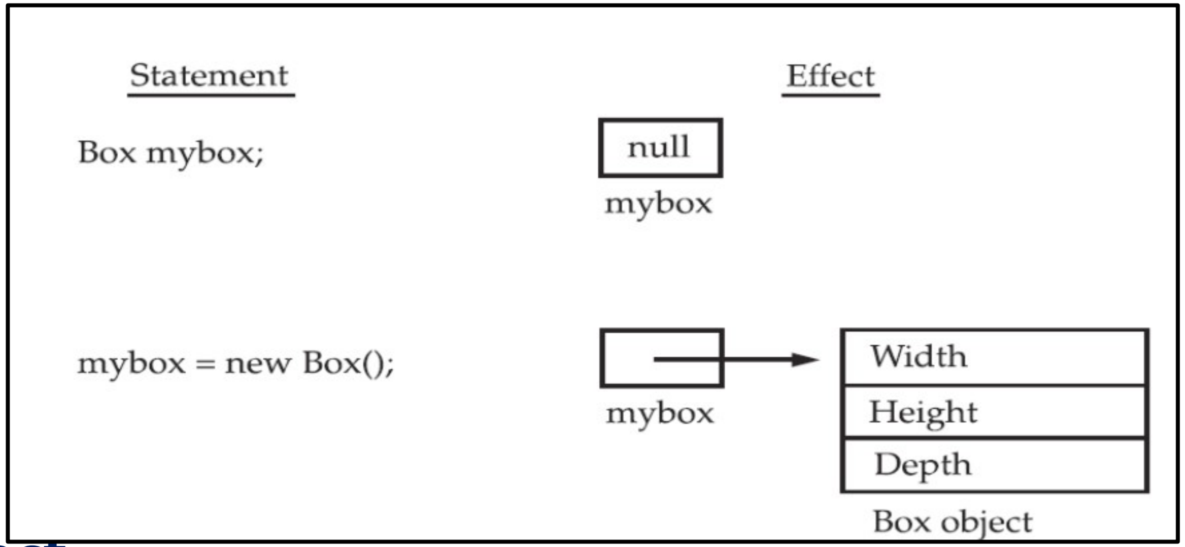
Classes and Objects

- The **general form of a class** is :

```
modifier class ClassName
{
      modifier type instance-variable1;
      modifier type instance-variable2;
      --------------------
       modifier type instance-variableN;
       modifier type methodname1(parameter-list)
      {
           // body of method
       }
       modifier type methodname2(parameter-list)
      {
           // body of method
       }
}
```

Classes and Objects

Reference variables can be assigned



Example-1:

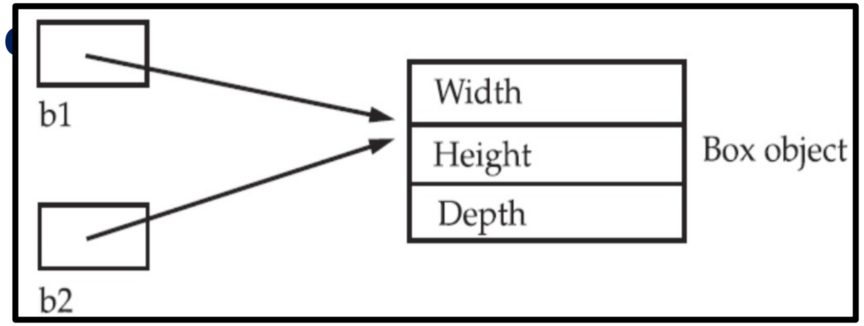
```
    Box mybox; //declaration
    mybox=new Box(); //Allocation
```

Example-2:

```
    Box b1 = new Box();
    Box b2 = b1; // b1, b2 refers to same object
```

Example-3:

```
    Box b1 = new Box() , b2 = new Box(); // both are
```



Example-4:

```
    Box b1 = new Box();
    Box b2 = new Box(); // both are different
```

Methods

- **type specifies** the type of **data returned** by the **method.**
- If the **method does not return a value**, its **return type must be void.**
- The **name of the method** is **any** valid **identifier** name.
- The **parameter-list** is a sequence of type and identifier pairs **separated by commas**.
- If the **method has no parameters**, then the parameter **list will be empty.**

Syntax

```
type
methodName(parameter-list)
{
      // body of method
}
```

Example

```
class Box
 {
        double width;
        double height;
        double depth;
        double volume()
        {
                  double x; // local variable
                                          x= width *
height * depth;
                 return x;
        }
        void setDim(double w, double h,
double d)
       {
      width = w;
      height = h;
      depth = d;
       }
 }
```

```
class BoxDemo
 {
      public static void main(String args[])
      {
            Box b1= new Box(); //b1 – reference variable
            Box b2 = new Box(); //b2-reference variable
            b1.setDim(10, 20, 15);
            b2.setDim(3, 6, 9);
            double vol1 = b1.volume();
            System.out.println(“box1 Volume is “ +vol1);
             System.out.println(“box1 Volume is “
+b2.volume());
 }
}
```

Example

Example-2

```
class Emp
{
      int id;
     String name;
     double salary;
     void insert(int i, String n, double s)
    {
          id=i;
         name=n;
         salary=s;
    }
    void display()
     {
          System.out.println(id+" "+name+"
"+salary);
      }
}
```

```
class TestEmp
 {
       public static void main(String[]
args)
       {
              Emp e1=new Emp();
              Emp e2=new Emp(),e3=new
Emp();
              e1.insert(101,"Varun",45000);

e2.insert(102,"Tharun",25000);
              e3.insert(103,"Nirun",55000);
              e1.display();
              e2.display();
              e3.display();
          }
 }
```

Forms of Methods

1. With No Parameters and No Return Value

```
void volume()
{
double x;
x= width * height * depth;
System.out.println(“The Volume is “ +x);
}
```

2. With Parameters and No Return Value

```
void volume(double len , double wid, double hgt)
{
double x=len*wid*hgt;
System.out.println(“The Volume is “ +x);
}
```

3. With Parameters and a Return Value

```
double volume(double len , double wid, double hgt)
  {
   double x= len*wid*hgt;
   return x;
  }
```

4. With No Parameters and a Return Value

```
double volume()
{
double x;
x= width * height * depth;
return x;
}
```

Constructors

- Java Allows **objects to be initialized themselves** when they are **created.**
- **Constructor** in java is a special type of method that is **used to initialize the object.**
- Java **constructor is invoked at the time of object creation**. It **provides data** for the **object** hence it is **known as constructor**.

Characteristics:

- Once defined, constructor is **automatically called after the object is created**,.
- Syntactically **similar to a method**
- The **name of the constructor must be** same as **its class name.**
- If **no constructor is defined**, a **default constructor** is **invoked by Java.**
- **Constructor must not** have **explicit return type**. By **default a constructor returns** the class **instance after creation.**

Rules for creating java constructor

There are basically two rules defined for the constructor.

i. **Constructor name must be same as its class name**

ii. **Constructor must have no explicit return type**

Types of constructors

- There are **two types of constructors:**

i. Default constructor (No-Argument constructor):

- Default constructor **provides the default values** to the object like 0 (int), 0.0(double), null (String) etc. **depending on the type.**
- The Constructor **has no arguments.**
- **Initializes all objects** to same, hence no longer used.

```
Box()
{
 Len = wid= hgt =0.0;
}
Eg: Box b1 = new Box();
 Box b2 = new Box();
```

//Example for default constructor-1

```
class Bike
{
     Bike()
     {
         System.out.println("Bike is
created");
     }
     public static void main(String args[])
      {
           Bike b=new Bike ();
      }
}
```

// Example for default constructor-2

```
class Box
{
      double len ;
      double wid;
      double hgt;
      Box()
      {
          Len = wid= hgt =0.0;
      }
      double volume( )
      {
          double res;
          res = len * wid * hgt;
          return res;
      }
      void display()
      {
            System.out.println("The Dimensions of box are");
            System.out.println("Length= "+len +"Width=" +wid + "Height="
+hgt);
      }
}
```

```
class const1
{
     public static void main(String args[])
     {
         Box b1 = new Box();
         b1.display();
          System.out.println("The volume of b1 is " +
b1.volume());
         Box b2 = new Box();
        b2.display();
        System.out.println("The volume of b2 is " +
b2.volume());
      }
}
```

ii. Parameterized constructor:

- A **constructor that has parameters** is known as **parameterized constructor.**
- **Used to create objects** of **different state** and **type.**
- Parameterized constructor is **used to provide different values** to the **distinct objects.**

```
 Box(double x)
{
 len = wid= hgt =x;
}
Box(double x ,double y , double z)
{
len = x;
wid = y;
hgt = z;
}
```

//Example for Parameterized constructor-1

```
class Student
 {
    int id;
    String name;
    Student (int i,String n)
    {
        id = i;
        name = n;
    }
    void display()
    {
       System.out.println(id+" "+name);
    }
    public static void main(String args[])
    {
        Student s1 = new Student (111,"Ramesh");
        Student s2 = new Student (222,"Suresh");
       s1.display(); s2.display();
    }
}
```

//Example for Parameterized constructor-2

```
class Box
{
	double len ;
	double wid;
	double hgt;
	Box(double x ,double y , double z)
	{
		len = x;
		wid = y;
		hgt = z;
	}
	double volume( )
	{
		double res;
		res = len * wid * hgt;
		return res;
	}
	void display()
	{
		System.out.println("The Dimensions of box
are");
		System.out.println(len +":" +wid + ":" +hgt);
	}
}
```

```
class const1
{
	public static void main(String args[])
	{
				 Box b1 = new Box(4,5,6);
				 b1.display();
				System.out.println("The volume of b1 is "
+ b1.volume());
				Box b2 = new Box(7.5,8.5,12.5);
				b2.display();
				System.out.println("The volume of b2 is "
+ b2.volume());
			}
}
```

iii.Copy constructor:

- There is **no copy constructor specific in java.** But, **we can copy** the values of **one object to another.**
- **Copy one object** into **another By constructor.**

ject into another.

```
class Student
{
      int id;
      String name;
      Student(int i,String n)
      {
            id = i;
            name = n;
      }
      Student(Student s)
      {
            id = s.id;
            name =s.name;
      }
      void display()
      {
          System.out.println(id+"
"+name);
      }
```

```
 public static void main(String args[])
 {
      Student s1 = new Student(1001,
“Varun”);
       Student s2 = new Student(s1);
      s1.display();
      s2.display();
 }
}
```

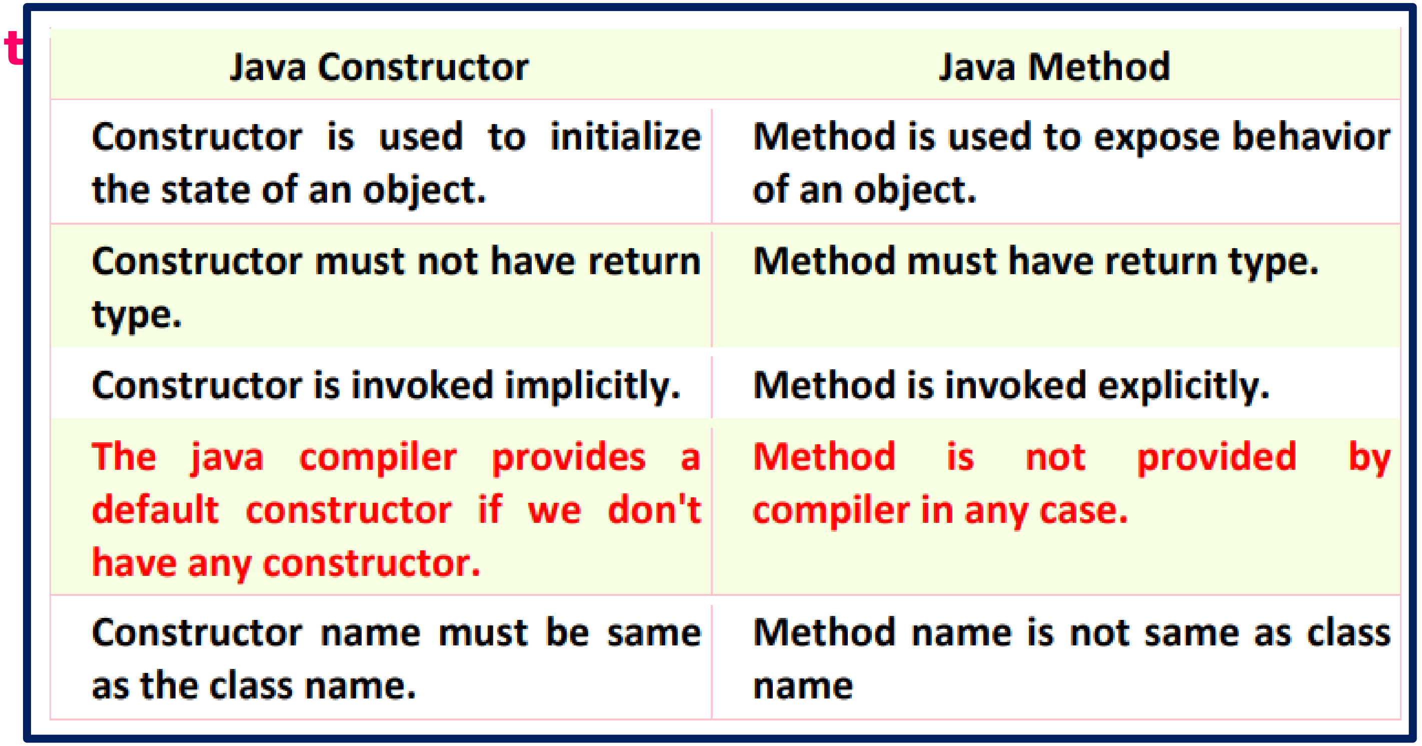
Difference between constructor and method in java

i. Does constructor return any value?

Yes, that is current class instance. (**We cannot** use **return type explicitly**, **it returns** an **object** by constructing it).

ii. Can constructor perform other tasks instead of initialization?

Yes, like **object creation, starting a thread, opening a file, calling method** etc. We **can perform any operat**



Java Constructor	Java Method
Constructor is used to initialize the state of an object.	Method is used to expose behavior of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if we don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name is not same as class name

this keyword

In java, **this is a reference variable** that **refers** to the **current object.**

Usage of this keyword

1. **this can be used** to **refer current class instance variable.**
2. this can be **used to invoke current class method** (implicitly)
3. this() can be **used to invoke current class constructor.**
4. this can be **passed as an argument** in the **method call.**

i. How ‘this’ refer current class instance variable:

- When a **local variable has the same name** as an **instance variable**, the local variable **hides** the **instance variable**. This is called **instance variable hiding**.(**Naming Collision**).
- we can use this keyword to **resolve any namespace collisions** that might occur **between instance** variables and **local variables.**

this keyword

Understanding the problem without this keyword

```
class Student
 {
        int rollno;
       String name;
       float fee;
       Student(int rollno,String name,float fee)
      {
          rollno=rollno;
          name=name;
          fee=fee;
       }
      void display()
     {
         System.out.println(rollno+" "+name+"
"+fee);
       }
 }
class TestThis
 {
        public static void main(String args[])
        {
             Student s1=new
Student(111,"ankit",5000f);
             Student s2=new
Student(112,"sumit",6000
            s1.display();
            s2.display();
        }
 }
```

Output

```
0 null 0.0
0 null 0.0
```

Solution of the above problem by this keyword

```
class Student
{
        int rollno;
        String name;
       float fee;
       Student(int rollno,String name,float fee)
       {
                this.rollno=rollno;     //resolve the problem of
instance variable hiding
                this.name=name;
                 this.fee=fee;
         }
       void display()
       {
              System.out.println(rollno+" "+name+" "+fee);
        }
}
class Test
{
        public static void main(String args[])
       {
       Student s1=new Student(111,"ankit",5000f);
       Student s2=new Student(112,"sumit",6000f);
        s1.display();
         s2.display();
         }
}
```

Output

```
111 ankit
5000.0
112 sumit
6000.0
```

ii. How ‘this’ invoke current class method:

- We may **call the method** of the **current class** by using this keyword.
- If we don't use the keyword, **compiler automatically adds this** keyword **while calling the method.**

```
class College
 {
      void VCE( )
      {
            System.out.println("hello VCE");
       }
      void IT( )
       {
            System.out.println("hello IT");
             this.hello();
       }
 }
class Test
 {
      public static void main(String args[])
      {
            College t1=new College();
            t1.IT();
      }
 }
```

Output

```
hello n
hello m
```

iii.How ‘this()’ invoke current class constructor:

- The this() constructor call can be used to **invoke the current class constructor**.
- It is **used to reuse the constructor**.
- In other words, it is **used for constructor chaining.**

```
class Student
 {
        int rollno;
        String name;
         float fee;
        Student(int r,String n)
        {
              rollno=r;
              name=n;
        }
        Student(int r,String n,float f)
        {
              this(r,n);                                  //reusing
constructor
               fee=f;
                          }
        void display()
        {
              System.out.println(rollno+" "+name+"
"+fee);
         }
}
```

```
class Test
 {
        public static void main(String args[])
       {
              Student s1=new Student(111,“Ram");
              Student s2=new Student(112,“Rishi",6000f);
              s1.display();
               s2.display();
        }
 }
```

Output

```
111 Ram  0.0
112 Rishi 6000.0
```

iv. How to pass ‘this’ as an argument in the method :

- The **this keyword** can **also be passed** as an **argument** in the method.
- It is **mainly used** in the **event handling**.
- Let's see the example:

```
class Test
{
     void VCE(Test obj)
      {
          System.out.println("method is invoked");
     }
     void IT()
      {
          VCE(this);
     }
      public static void main(String args[])
     {
          Test t1 = new Test();
          t1.IT();
     }
}
```

Output

```
method is invoked
```

static Keyword

- If we **apply static keyword with any method then the method is called as** static method.
- The restrictions of static on methods declared is:
  1. They can **directly call other static methods.**
  2. They can **directly access static data.**
  3. They **cannot refer to with “this”** or **“super” keys** in a
- A **static method or variable belongs** to **a class** rather than **object.**
- A static method can be **invoked without** the **need for creating an object of a class.**
- **static method cannot access non static variables** or **non stat**
- It is **used to initialize** or **modify** the **static data.**
- It **can be accessed directly** using **ClassName.**

Syntax to access Static Variable

ClassName.Variable_Name;

Syntax to access Static Method

ClassName.methodName();

Why Java main() method is static?

Java main method is static because

i) An **object is not required to call** static method.

ii) If it is **non static method, JVM creates object first then call main( )** this leads to

static variable example

//Program for static variable

```
class Student
{
   int rno;                              //instance
variable
   String name;
   static String college ="VCE";  //static
variable

   Student(int r, String n)
   {
     rno=r;
     name=n;
   }
    void display ()
    {
          System.out.println(rno+" "+name+"
"+college);
    }
}
```

```
class TestStudent
{
      public static void main(String args[])
      {
      Student s1 = new
Student(1201,"Akhil");
      Student s2 = new
Student(1202,"Avinash");
      s1.display();
      s2.display();
      Student.college="NIT"
       s1.display();
      s2.display();
       }
}
```

Output

```
1201        Akhil     VCE
1202      Avinash    VCE
1201        Akhil      NIT
1202      Avinash     NIT
```

static variable and instance variable Difference

//Program for instance variable

```
class Test
{
 int count=0;

Test()
{
count++;
System.out.println(count);
}

public static void main(String args[])
{
Test t1=new Test();
Test t2=new Test();
Test t3=new Test();
}
}
```

Output

```
1
1
1
```

//Program for static variable

```
class Test
{
static int count=0;

Test()
{
count++;
System.out.println(count);
}

public static void main(String args[])
{
Test t1=new Test();
Test t2=new Test();
Test t3=new Test();
}
}
```

Output

```
1
2
3
```

static method

```
//static method example -1
class Test
 {
        static int cube(int x)
        {
              return x*x*x;
        }
        static void display()
        {
               int y ;
               y = cube(6);
               System.out.println(y);
        }
       }
       class StaticDemo1
       {
               public static void main(String args[])
               {
               int result=Test.cube(5); //Directly using
              class name
               System.out.println(r
              Test.display();
        }
}
```

Output

```
125
216
```

Method Overloading

- **Polymorphism** in Java is a concept by which we **can perform a single action** in **different ways**. i.e **Poly means** "**many**" and **morphs** means "**forms**".
- The two type of Polymorphism
    1. **Compile Time polymorphism**
    2. **Run Time Polymorphism**
- **Compile Time polymorphism** in Java is **achieved** using "**method overloading**".
- **Defining two** or **more methods within** the **same class** that **share the same name** but **parameter** declarations **are different.** The **methods are** said to be **overloaded** and **the process** is referred as **method overloading.**
- When an **overloaded method** is **invoked** uses **type** and **number of arguments** to **determine which version** is **called.**
- Overloaded methods **must differ in type** and/or **number of parameters.**
- When an overloaded method is called, **Java simply executes** the version of the **method whose parameters match** the **arguments used in the call.**

Method Overloading

Advantage of method overloading

i. Method **overloading increases** the **readability** of the **program.**

ii. Provides **Compile time Polymorphism**

- Different ways to overload the method
- There are **two ways to overload the method** in java

    i. By changing number of arguments

    ii. By changing the data type

i.Method Overloading: By changing number of arguments

```
class Adder
 {
     int add(int a,int b)
     {
            return a+b;
     }
      int add(int a,int b,int c)
     {
           return a+b+c;
     }
 }
class Test
 {
     public static void main(String[] args)
     {
          Adder a = new Adder();
          System.out.println(a.add(10,11));            //2-args
Version
          System.out.println(a.add(10,11,12));      //3-args
Version
     }
 }
```

Note

When a overloaded method is called, Java **executes the** version of **the method whose parameters match** the arguments used in the **call.**

ii. Method Overloading: changing data type of arguments

```
class Adder
 {
      int add(int a, int b)
      {
           return a+b;
     }
     double add(double a, double b)
     {
          return a+b;
     }
 }
class Test
 {
     public static void main(String[] args)
     {
          Adder a = new Adder();
          System.out.println(a.add(11,11));               //int version
          System.out.println(a.add(12.3,12.6));         //double
version
     }
 }
```

Overloading Constructors

- Constructor overloading is a technique in which a **class can have any number of constructors** that **differ in parameter lists.**
- The **compiler differentiates** these constructors **by taking** into account the **number of parameters** in the list and their data type.
- Like methods, the **constructors in java can also be overloaded**.
- This **process is called constructor overloading.**

Overloading Constructors

```
//Example of Constructor
Overloading -1
 class Student
 {
      int id;
      String name;
      int age;
      Student (int i,String n)
      {
      id = i;
      name = n;
      }
      Student (int i,String n,int a)
      {
      id = i;
      name = n;
      age=a;
      }
```

```
void display()
 {
  System.out.println(id+" "+name+"
"+age);
  }

 public static void main(String args[])
 {
      Student s1 = new
     Student(111,"Karthik");
      Student s2 = new
     Student(222,"Aryan",25);
       s1.display();
      s2.display();
     }
}
```

Parameter Passing in Java

1. Pass By Value:

- **Any modifications** to the **formal parameter** variable inside the called method affect only the separate storage location and **will not be reflected** in the **actual parameter** in the calling environment.

ts and arrays, the copy semantics are costly

```
//Parameter Passing Program - Call By Value
 class Test
{
 public static void update(int x, int y)
 {
 x++;
 y++;
 System.out.println("x,y values are" +x + " ,
" +y);//11,21
 }
}
```

```
class ByVal
{
     public static void main(String[] args)
     {
        int a = 10;
         int b = 20;
        Test t = new Test();
        System.out.println("The values of a, b before
method call");
        System.out.println("a, b values are" + a + ","
+ b); //10,20
         t.update(a, b);
         System.out.println("The values of a, b after
method call");
         System.out.println("a, b values are" + a + ","
+ b); //10,20
     }
```

Parameter Passing in Java

2. Call by reference(aliasing):

- **Any changes** to the **formal parameter** are **reflected** in the **actual parameter** in the calling environment as formal .
- This method is **efficient in both time and space.**

```
class Demo
{
 int a, b;
 void update(Demo obj)
 {
      obj.a += 10;
      obj.b += 20;
 } }
class ByRef
{
      public static void main(String[] args)
      {
            Demo d = new Demo();
           d.a = 10;
           d.b = 20;
           System.out.println("a, b before method call");
           System.out.println( d.a + "," + d.b);
    //10,20
           d.update(d); // Method Call
           System.out.println("a, b after method call");
           System.out.println("a, b values are" + d.a + "," + d.b);
 //20,40
      }
}
```

String Class

- String is a **sequence of characters**. But in Java, **string is an object** that **represents a sequence of characters.**
- The **java.lang.String class** is **used to create** a **string object.**
- Java String class **provides methods to perform operations on string**
- A String **object is created using**:

By String Literal

- Java **String literal** is **created** by using **double quotes.**

```
String s="welcome";
```

- Each time you create a string literal, the **JVM checks the "string constant pool"** first. **If the string already exists** in the **pool**, a **reference** to the pooled **instance is returned**.
- If the **string doesn't exist** in the **pool**, a **new string instance** is **created** and placed in the pool.
- To make Java more memory efficient (because **no new objects are created if it exists already** in the **string constant pool).**

```
String s1="Welcome";
```

String Class

By new keyword.

- String class **supports several constructors** to **create strings.**

1. String s = new String( ); **//default constructor – empty string**

2. String s=new String("Welcome");

3. String(char ch[ ]); **// parameterized constructor - char array to string object**

```
    char ch[] = { 'w', 'e', 'l',’c’,’o’,’m’,’e’ };
    String s1 = new String(ch); //”welcome”
```

4. String(char ch[ ], int startIndex, int numChars) **// parameterized constructor – char array to string object**

```
    char ch[] = { 'w', 'e', 'l',’c’,’o’,’m’,’e’ };
    String s2 = new String(ch, 3,4); //”come”
```

5. String(String strObj) **// copy constructor**

```
    String s1 = new String(“Hyderabad”);
    String s3 = new String(s1);
```

String Class

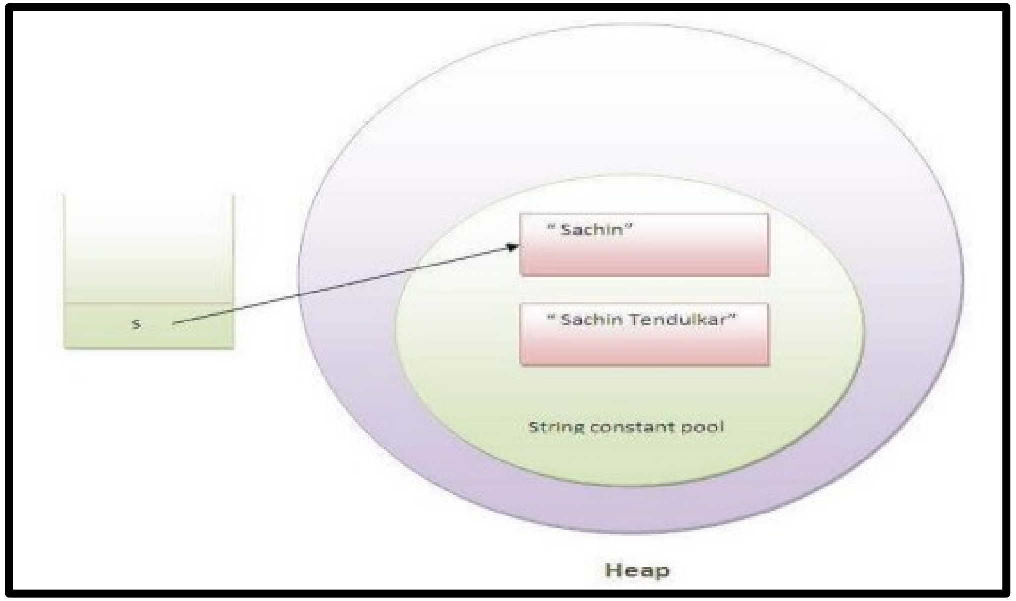
6. String(byte asciiChars[ ])

```
String(byte ascii[ ])
byte ascii[] = {65, 66, 67, 68, 69, 70 };
String s1 = new String(ascii); //ABCDEF
System.out.println(s1); //”ABCDEF”
```

- The Java **String is immutable** which **means it cannot be changed.**
- Whenever **we change any string**, a **new instance is created.**
- Once **string object is created its data** or state **can't be changed** but a **new string object** is **created with changes.**

String Class

```
class String1
{
      public static void main(String args[])
       {
      String s="Sachin";
      s.concat(" Tendulkar");
      System.out.println(s); // Sachin
       }
}
```



- Here **Sachin** is **not changed** but a **new object** is **created with Sachin Tendulkar.**
- **That is why string** is known as **immutable.**
- As you can see in the above figure that **two objects are created but s reference variable still** refers to **"Sachin"** not to **"Sachin Tendulkar".**
- But **if we explicitly assign** it to the **reference variable,** it will **refer to "Sachin Tendulkar"** object.

String Class

```
class String1
{
     public static void main(String args[])
      {
      String s="Sachin";
      s1= s.concat(" Tendulkar");
      System.out.println(s1);
      }
     }
     Output: SachinTendulkar
```

Reason:

- Suppose there are **5 reference variables**, all **refers to one object "sachin".**
- If **one reference variable changes** the value of the object**, it will be affected to all the reference variables.**
- **That is why string objects** are **immutable** in java.
- **For mutable strings**, we can **use String Buffer** and **StringBuilder classes** are used.

Methods of String Class

Method Name	Description	Example

int length()	returns string length	**String s1="java";**
		System.out.println("string length is "+s1.length()); //4
		System.out.println("string length is: "+"java".length()); //4

char charAt(int index)	Extracts a single character from the string. Returns char value for the particular index.	**String str="welcome";** **char ch=str.charAt(3); //c**

String substring(int beginIndex)
String substring(int beginIndex, int endIndex)
	Used to extract a substring from specified index from the main string. Returns a substring.	**String s1="welcome";**
		String str1= s1.substring(3); //come
		String str2 = s1.substring(3,5); //com

String toLowerCase()	Returns a string in lowercase.	**String s1="OBJECT oreinTED";**
		String str=s1.toLowerCase();

Methods of String Class

Method Name	Description	Example
getChars (int start, int end, char target[] , int tarstart)	Used to extract more than one character at a time.	**Char buf[ ] = new char[10];** **String str= ”Welcome to Hyderabad”;** **str.getChars(3,6,buf,0) //come**

boolean equals(Object str)	Returns true if the strings contain same characters in same order otherwise returns false.	**String s1=”hello”;**
		String s2=”hello”;
		String s3=”HELLO”;
		System.out.println(s1.equals(s2)); //true
		System.out.println(s1.equals(s3)); //false

boolean equalsIgnoreCase(String str)	Compares two strings for equality and ignores the case differences.	**boolean x=s1.equalsIgnoreCase(s3)); //true**
boolean startsWith()	Returns true, if the given string	**System.out.println(“welcome”.startsWith(“wel”));//true**

Methods of String Class

Method Name	Description	Example
String concat(String str)	**The two strings are concatenated.**	**String s1 = “Welcome”;** **String s2 = s1.concat(“To India”);**
String replace(char old , char new)	**Replaces all occurrences of one character in the invoking string with another character.**	**String s1 = “Welcome”;** **String s2 = s1.replace(‘e’ , ‘o’);**
String trim()	**Returns a copy of invoking string from which any leading and trailing white space has been removed.**	**String s1 = “ Welcome ”;** **String s2 = s1.trim();** **//”welcome”**
boolean isEmpty()	Returns true if the string object is empty	**String s=””;** **boolean x = s.isEmpty();** **//true**
String[] split(String exp)	Returns a split string matching regular expression.	**String str=”Welcome to Hyd for biryani”;** **String s[] = str.split(“ ”); //splits string with white spaces.**

Methods of String Class

Method Name	Description	Example

int compareTo(String str)	Used to compare to strings. It returns < 0: The invoking string is less than the given string. >0: The invoking string is greater than the given string. 0: if the two strings are equal.	String s1 =”welcome”;
		String s2= “hello”;
		String s3 = “welcome”;
		System.out.println(s1.compareTo(s3)); //0
		System.out.println(s1.compareTo(s2)); // >0
		System.out.println(s2.compareTo(s3)); // <0

Searching Strings
int indexOf(int ch)
int indexOf(String str)
int indexOf(int ch , int start)
	Used to search a string for a specified character or substring. Returns index at which the character or string is found , Otherwise returns -1.	String str = “to do welcome to indian railways to come and well do of the rail works”
		int x = str.indexOf(‘w’);
		int x = str.indexOf(“wel”);
		int x = str.indexOf(‘w’ , 10);

String Buffer Class

- StringBuffer class is **used to create a mutable string object**. It means, it **can be changed after it is created**.
- It is **similar to String class** but **stringbuffer class object can be changed**.
- **Creating a StringBuffer Object:**

i. StringBuffer(): It **creates** an **empty string buffer** and **reserves space for 16 characters.**

```
StringBuffer  <object_name> = new StringBuffer();
```

ii. StringBuffer(int capacity): It is **used** for initializing the **initial size of the StringBuffer**. This constructor takes one parameter i.e. the capacity of the StringBuffer.

StringBuffer ob = new StringBuffer(15); //Initialising the **StringBuffer capacity to the 15 bytes**

iii.StringBuffer(String str): It accepts the String as a parameter and **converts the String** to the **StringBuffer object.**

String Buffer Methods

- **StringBuffer class provides various types** of **inbuilt methods** that help to **perform different operations** on Strings.

1. int length():

It returns the number of characters present in SB

```
class Test
{
	public static void main(String[] args)
	{
	StringBuffer sb = new
StringBuffer("abc");
	System.out.println(sb.length());
	}
}
```

Output:3

2. int capacity():

It returns the max number of characters it can store

```
class Test
{
	public static void main(String[] args)
	{
	StringBuffer sb = new
StringBuffer("abc");
	System.out.println(sb.length());//3
	System.out.println(sb.capacity());//
16+3=19
	}
}
```

```
Output:3
       19
```

String Buffer Methods

3. setLength(int)

It change the length of SB

```
class Test
{
      public static void main(String[] args)
      {
              StringBuffer sb = new
StringBuffer("welcome");
      System.out.println(sb);//welcome
      System.out.println(sb.length());//7
      sb.setLength(2);
      System.out.println(sb);//we
      System.out.println(sb.length());//2
      }
}
```

```
Output: welcome
7
We
2
```

4. charAt(int index)

it returns the character located at specified location

```
class Test
{
      public static void main(String[] args)
      {
      StringBuffer sb = new
StringBuffer(“Vardhaman");
      System.out.println(sb.charAt(0));//V
      }
}
```

```
Output: V
```

String Buffer Methods

5. setChar(int index,char ch)

it is used to inserts the char at index

```
class Test
{
      public static void main(String[] args)
      {
                   StringBuffer sb = new
StringBuffer(“JVA");
        System.out.println(sb);//JVA
        sb.setCharAt(1,’A');
         System.out.println(sb);//JAVA
      }
}
```

```
Output:JVA
JAVA
```

6. deleteCharAt(int index)

it removes char located at index

```
class Test
{
      public static void main(String[] args)
      {
                   StringBuffer sb = new
StringBuffer("welcoome");
        System.out.println(sb);//welcoome
        sb.deleteCharAt(4);
        System.out.println(sb);//welcome
      }
}
```

```
Output: welcoome
```

String Buffer Methods

7. append(String)

it insert the arg at the end of sb

```
class Test
{
	public static void main(String[] args)
	{
	    StringBuffer sb = new StringBuffer();
	    System.out.println(sb);//
	    sb.append("welcome");
	    sb.append(" ");
	    sb.append("java");
	    System.out.println(sb);//welcome java
	}
}
```

```
Output:
welcome java
```

8.insert(int index,String)

it inserts the string at specified location

```
class Test
{
	public static void main(String[] args)
	{
			StringBuffer sb = new
StringBuffer("welcome java");
			System.out.println(sb);//welcome java
	sb.insert(8,"to");//welcome tojava
	sb.insert(10," ");
	System.out.println(sb);//welcome to java
	}
}
```

```
Output: welcome java
welcome tojava
welcome to java
```

String Buffer Methods

9.delete(int start,int end)

it removes char from start index to end index

```
class Test
{
	public static void main(String[] args)
	{
	StringBuffer sb = new StringBuffer("welcome to java");
	sb.delete(7,10); //start to end-1
	System.out.println(sb);//welcomejava
	}
}
```

Output:welcomejava

10.reverse()

it reverse the given String

```
class Test
{
	public static void main(String[] args)
	{
	StringBuffer sb = new StringBuffer(“VCE");
	sb.reverse();
	System.out.println(sb);//CEV
	}
}
```

Output:CEV

StringTokenizer Class

- **StringTokenizer** is **used to break a string into tokens based** on provided **delimiter.**
- It is present in **java.util package**.
- we are using **the constructor** of this class to **specify the input string** and **delimiter:**

  i.StringTokenizer(String str):

  It takes **string as argument** and uses the **default delimiter as ” \t\n”:** the **space character, the tab character, the newline character** etc.

  ii.StringTokenizer(String str, String delim):

  creates a string tokenizer for the **specified string** by using **specified delimiter**. Delimiter character will be treated as separator.

- Methods of StringTokenizer class:

  i. **boolean hasMoreTokens():** Returns **true if the next token** is **available.**

  ii. **String nextToken():** It **returns the next token.**

StringTokenizer Class

i.StringTokenizer(String str)

```
import java.util.StringTokenizer;
class Test
{
public static void main(String[] args)
{
        String s = new String("welcome to
java");
        StringTokenizer st = new
StringTokenizer(s);

System.out.println(st.countTokens());//3
}
}
```

Output: 3

ii.StringTokenizer(String str, String delim)

```
import java.util.StringTokenizer;
class Test
{
public static void main(String[] args)
{
        String s = new String("10,20,30,40,50");
        StringTokenizer st = new
StringTokenizer(s,",");
        System.out.println(st.countTokens());//5
}
}
```

Output: 5

StringTokenizer Class

hasMoreTokens() method:

```
import java.util.StringTokenizer;
class Test
{
    public static void main(String[] args)
    {
        String s = new String("welcome to java ");
        StringTokenizer st = new StringTokenizer(s);
        while(st.hasMoreTokens())
        {
            System.out.println(st.nextToken());
        }
    }
}
```

Output:

```
welcome
to
java
```

Date Class

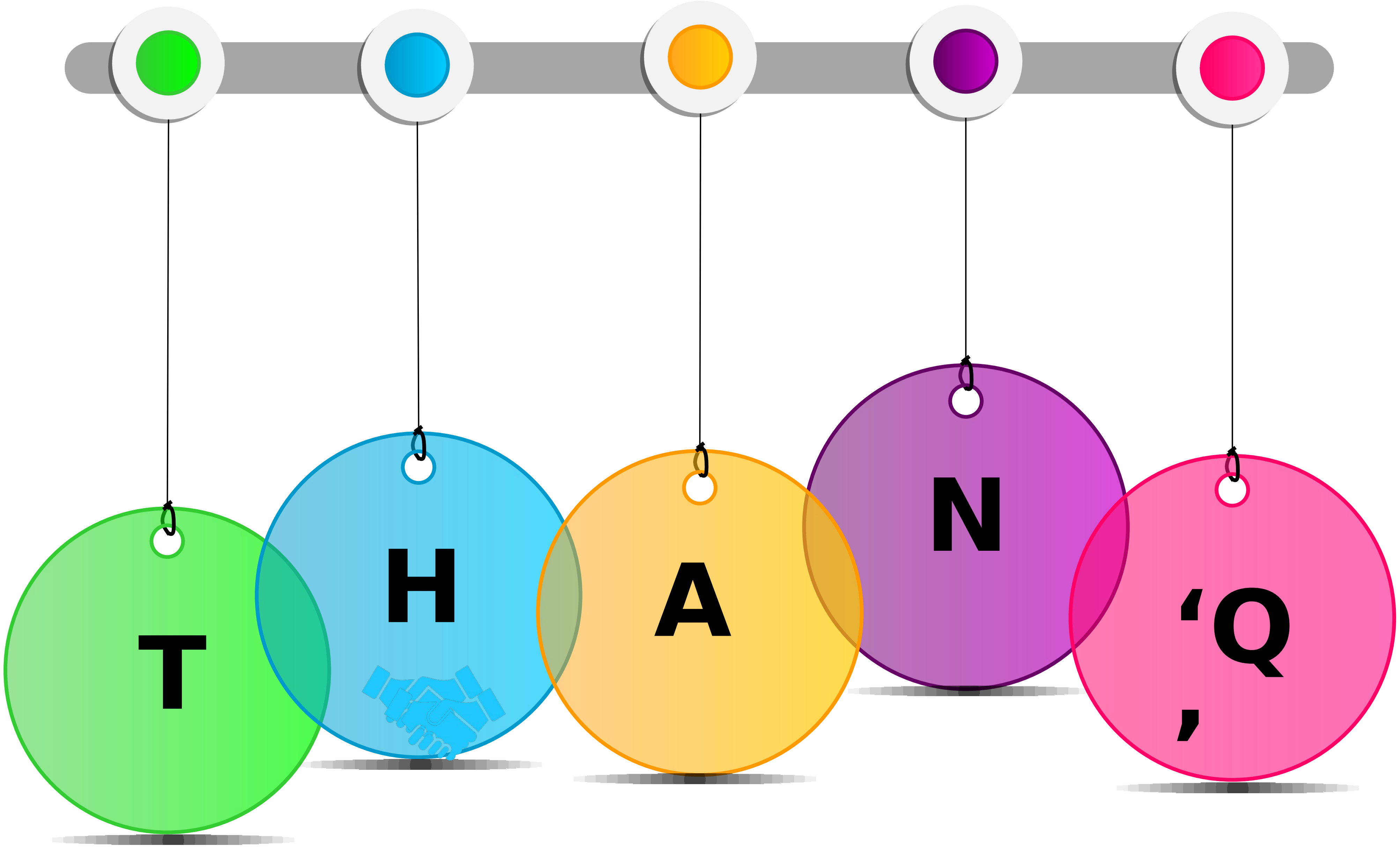
- The **java.util.Date class represents date and time in java**.
- It **provides constructors** and **methods** to deal with date and time in java.
  - i. Date() : **Creates date object** representing **current date and time.**
  - ii. Date(long milliseconds) : **Creates a date object** for the **given milliseconds since January 1, 1970**, 00:00:00 GMT.
- Other Important methods of Date class:
  - **i. boolean after(Date date) :** Tests if **current date** is **after** the given **date.**
  - **ii. boolean before(Date date) :** Tests if **current date** is **before** the given **date.**
  - **iii. int compareTo(Date date) :** Compares **current date** with **given date**. **Returns 0** if the argument **Date is equal to the Date**; a value **less than 0 if the Date is before** the Date argument; and a value **greater than 0** if the Date is **after the Date** argument.

Date Class

```
//Example of data class in java
// Java program to demonstrate constuctors of Date
import java.util.*;

public class Main
{
	public static void main(String[] args)
	{
		Date d1 = new Date();
		System.out.println("Current date is " + d1);
		Date d2 = new Date(2323223232L);
		System.out.println("Date represented is "+ d2 );
	}
}
```

Output

```
Current date is Sun Nov 12 18:35:37 IST
2023
Date represented is Wed Jan 28 02:50:23
IST 1970
```



T

H

A

N

‘Q

,